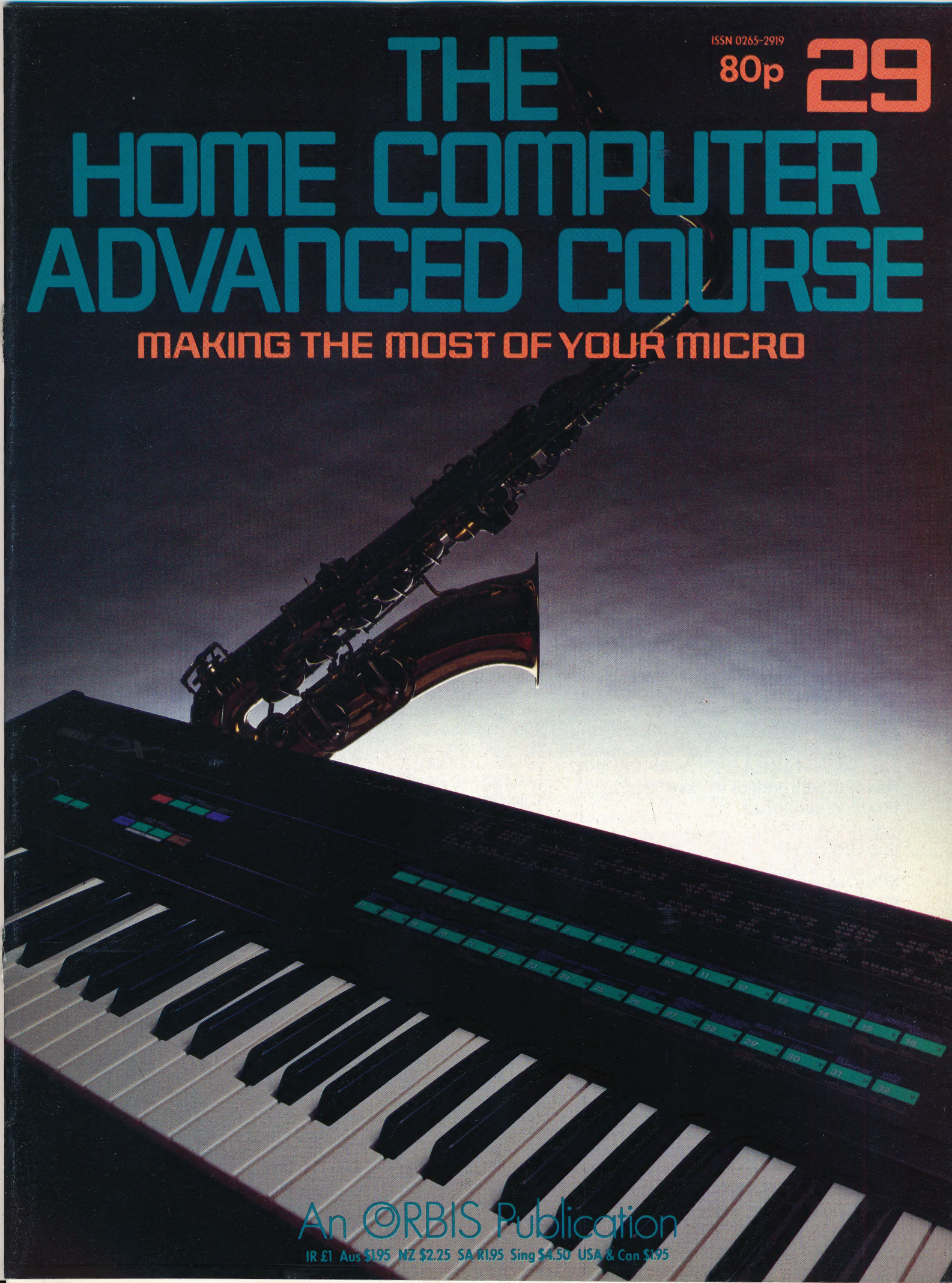
ISSN 0265-2919

80p

29

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



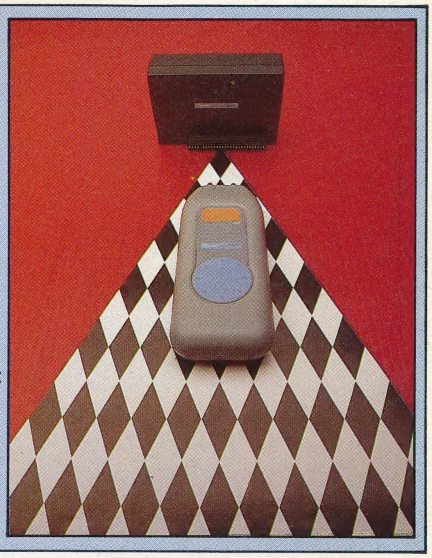
An ORBIS Publication

IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

Next Week

- **The Cheetah RAT (Remote Action Transmitter) is a recent addition to the growing range of infra-red peripherals. We examine the first wire-less joystick**
- **We continue our programming projects with the Spectrum and BBC Micro versions of the user-defined character generator program**
- **In the concluding instalment of our music series we survey some of the more advanced electronic instruments and techniques currently in use**



QUIZ

1) What does 'state transparent' mean?
2) How is PRODOS unusual among operating systems?
3) Why is MIDI advantageous?
4) What was Memotech's first product?

Answers To Last Week's Quiz

1) The UNIX manual is completely implemented as on-line help pages in the system itself.

2) The disadvantage of MIDI's seriality is the polyphonic deterioration introduced by transmission delay.

3) The 6809 is unusual firstly, in having two eight-bit accumulators, and, secondly, in combining them as a single 16-bit register.

4) A copy of J.R.R. Tolkien's novel, 'The Hobbit', was given away with every copy of Melbourne House's game of the same name, at the insistence of the Tolkien estate.

Editor Jim Lennox; **Managing Editor** Mike Wesley; **Art Director** David Whelan; **Technical Editor** Brian Morris; **Production Editor** Catherine Cardwell; **Art Editor** Claudia Zeff; **Chief Sub Editor** Robert Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Editorial Assistant** Stephen Malone; **Sub Editor** Steve Mann; **Researcher** Melanie Davis; **Contributors** Geoff Bains, Harvey Mellor, Mike Curtis, Steve Colwill, Steve Malone, Rory Forsyth, Graham Storrs; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editors** Maurice Geller, Chris Cooper; **Production Controller** Peter Taylor-Medhurst; **Circulation Director** David Breed; **Marketing Director** Michael Joyce; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOME COMPUTER ADVANCED COURSE – Price UK 80p IR £1.00 AUS $1.95 NZ $2.25 SA R1.95 SINGAPORE $4.50 USA and CANADA $1.95

How to obtain your copies of HOME COMPUTER ADVANCED COURSE – Copies are obtainable by placing a regular order at your newsagent, or by taking out a subscription. Subscription rates: for six months (26 issues) £23.80; for one year (52 issues) £47.60. Send your order and remittance to Punch Subscription Services, Watling Street, Bletchley, Milton Keynes, Bucks MK2 2BW, being sure to state the number of the first issue required.

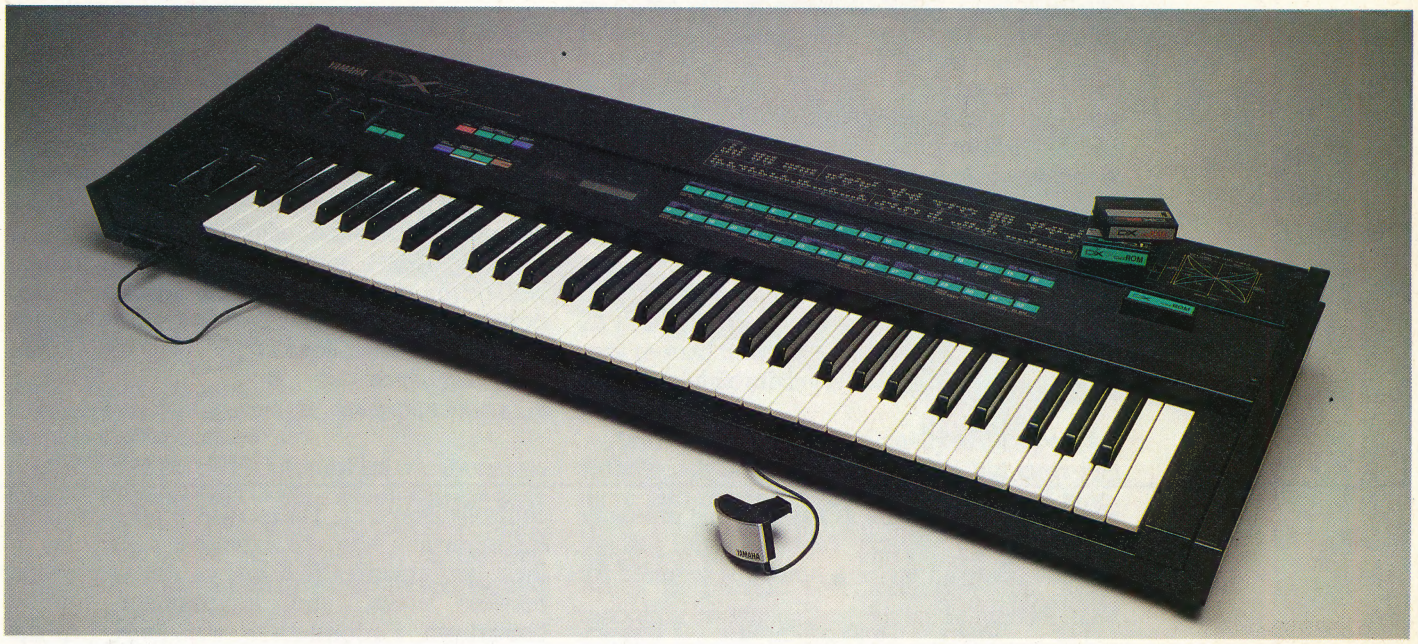
Back Numbers UK and Eire – Back numbers are obtainable from your newsagent or from HOME COMPUTER ADVANCED COURSE. Back numbers, Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT at cover price. AUSTRALIA: Back numbers are obtainable from HOME COMPUTER ADVANCED COURSE. Back numbers, Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G Melbourne, Vic 3001. SOUTH AFRICA, NEW ZEALAND, EUROPE & MALTA: Back numbers are available at cover price from your newsagent. In case of difficulty write to the address in your country given for binders. South African readers should add sales tax.

How to obtain binders for HOME COMPUTER ADVANCED COURSE – UK and Eire: Please send £3.95 per binder if you do not wish to take advantage of our special offer detailed in Issues 5, 6 and 7. EUROPE: Write with remittance of £5.00 per binder (incl. p&p) payable to Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT. MALTA: Binders are obtainable through your local newsagent price £3.95. In case of difficulty write to HOME COMPUTER ADVANCED COURSE BINDERS, Miller (Malta) Ltd, M.A. Vassalli Street, Valletta, Malta. AUSTRALIA: For details of how to obtain your binders see inserts in early issues or write to HOME COMPUTER ADVANCED COURSE BINDERS, First Post Pty Ltd, 23 Chandos Street, St. Leonards, NSW 2065. The binders supplied are those illustrated in the magazine. NEW ZEALAND: Binders are available through your local newsagent or from HOME COMPUTER ADVANCED COURSE BINDERS, Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington. SOUTH AFRICA: Binders are available through any branch of Central Newsagency. In case of difficulty write to HOME COMPUTER ADVANCED COURSE BINDERS, Intermag, PO Box 57394, Springfield 2137.

Note – Binders and back numbers are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK market only and may not necessarily be identical to binders produced for sale outside the UK. Binders and issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.

COVER PHOTOGRAPHY BY MARCUS WILSON-SMITH

SYNTHETIC FIBRE



MARCUS WILSON-SMITH

New Sounds, New Style
One of several new synthesisers in the DX range from Yamaha, the DX7 incorporates a method of building sound, FM synthesis, previously restricted to machines costing thousands of pounds. Instead of taking an existing sound and modifying it by passing it through filters or adjusting envelope controls, the DX7 creates it own complex sounds by combining six waveforms in a variety of ways. As a result, the DX7 approximates the sounds of acoustic instruments much more closely than other synthesisers. The DX7 also incorporates breath control, so a musician can blow into a receiver and add breath-like variations to saxophone or trumpet sounds, for example. The DX7 can use ROM packs with prerecorded sound characteristics, or it can store sounds you create on RAM packs. The DX7 sells for about £1,400

Of all the available microcomputer-based music systems, MIDI offers the widest range of applications. Musicians use the interface to upgrade performance capabilities, both live and in the studio. And MIDI offers the home computer owner a way into the world of electronic music.

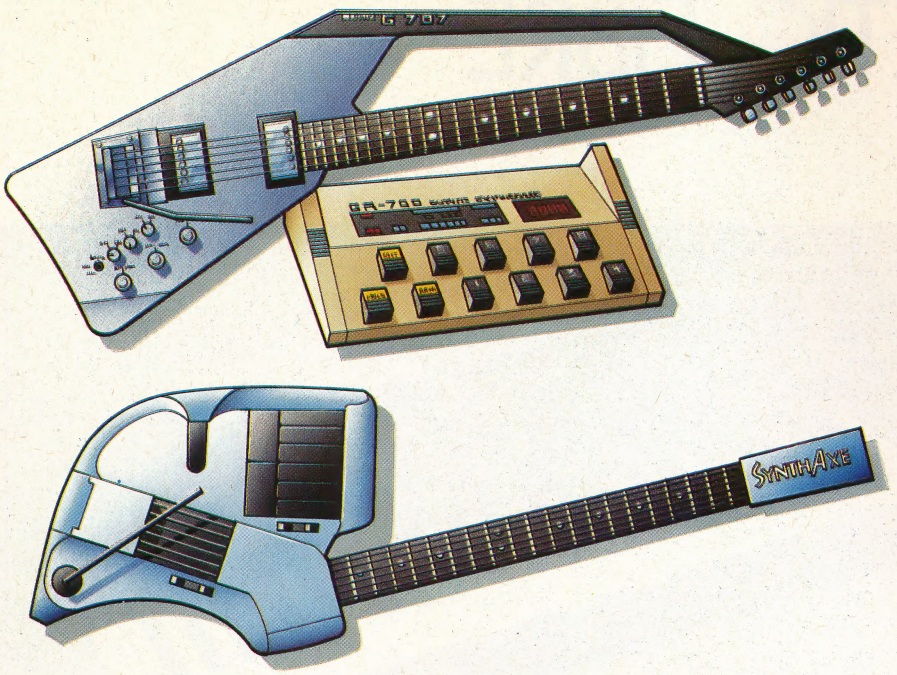
In education, MIDI offers some very real advances, both in schools and in the home. Learning to read music is often difficult, even when the learner is already quite competent as a performer. The early stages of working out such things as sharps and flats and time signatures can be laborious and time-consuming, and the connection between the marks on the paper and what is actually heard rarely seems obvious.

A major part of the problem lies in the nature of music itself: in order to make any sense, it has to be a series of events occurring through time. If the only way that a beginner can keep track of the visual notation is to keep stopping the music in question, then all the time-duration indicators in the score become pointless. Similarly, the beginner may spend some time in trying to interpret a particular bar or sequence; while this is being done the music will have moved on to something quite different.

MIDI removes these obstacles. It can enable a synthesiser performance to be stored in a micro's memory and, with the appropriate software, will give a graphic display of the music being played. This is an invaluable aid to any music student — it means that if, say, a Middle C is played on the synthesiser keyboard, then a Middle C will be shown on the five-line stave on the display screen. If a B minor chord is held down for a certain length of time, the harmonic components of the chord — B, D and F — will be displayed, together with the appropriate duration.

This idea could be extended by software that plays a pre-programmed piece of music on the interfaced synthesiser while a fully notated score rolls by on the screen. In this situation, both the music and its notation could be stopped simultaneously and re-run from a specified bar number if the user encounters a problem. In addition, the overall sound of the music could be varied by changing the control parameters on the synthesiser — thus introducing the user to the art of arranging.

Once a degree of confidence in reading music has been attained, then it should be easier to write music with the aid of the alphanumeric computer keyboard. This may involve entering performance data without any immediate reference to a sound from the synthesiser and then testing the result against the original intentions. Once this more advanced skill is acquired, five-line stave notation could be discarded in favour of another system such as MCL (music composition language). For electronic music, an MCL is a more appropriate means of entering data as it includes specification of characteristics exclusive to electronic sound production. No standard has been developed for MCL application — each machine has its own MCL. Stave notation, although a useful



KEVIN JONES

Synthesised Sound

Recently, several manufacturers have developed synthesisers that are activated from the strings of a guitar rather than the keys of a piano. The Roland GR700 uses standard guitar strings. Complex sound information is picked up by a hexaphonic (six-sound) input and sent to the synthesiser where the non-guitar parameters are added. A more exotic approach to the same technique is the SynthAxe, a 6809-controlled synthesiser. The SynthAxe, a development prototype, picks up sound data from an electrical connection between the string and the fret. Sensors at the top of the neck pick up bending and sliding variations. A second set of strings and a small keyboard are used to further enhance the sound

immediate visual guide, is a system that pre-dates electronics by several centuries, and cannot accommodate all the new notations required for electronic music.

For many microcomputer owners, understanding stave notation or MCL will be the key to getting the most from MIDI. For students in schools and colleges, a major obstacle is the present character of music education. For most music students, the accepted area of study is still firmly based in European classical music. The majority of classical musicians and teachers identify electronic music with the more avant-garde or radical composers of the last two decades, and, to an extent, with contemporary pop music. Neither of these fields is accepted as being on a par with classical music. In fact, some classicists would even question whether they were 'music' at all.

It seems unlikely, therefore, that MIDI's educational potential will be much explored in the mainstream music curriculum, especially since computing skills are required in addition to musical electronics. There can be few computer science courses with sound-proofing facilities that are adequate for a class of MIDI synthesiser players. If MIDI proves to be popular in such an environment, a simple solution such as the use of headphones will be required. But, as most MIDI units are designed to interface with one or more synthesisers, several signals would be required to deal with this. Even at a basic level, MIDI use in education implies the development of computer music studio facilities, and it would seem to require an active involvement by computer science students as well as by music pupils.

For live performances, MIDI is primarily a means of integrating a number of synthesisers, sequencers and drum machines into a single controllable system. The most worrying possibility for musicians who use extensive sequencing in their performances is a loss of synchronisation between units, and a resultant musical breakdown. Performers as well known as the Thompson Twins and Howard Jones have been known to run tapes of their studio-recorded backing tracks while 'performing live', rather than take the risk. In theory at least, multi-synthesiser groups should have more trouble-free performances thanks to MIDI.

One feature of this advance is that such groups will no longer have the appearance of being 'multi-synthesiser'. Since the early 1970s, a synthesiser has generally been thought of as a keyboard instrument, with a number of parameter control knobs and sliders set in a fascia above the keys. But if one keyboard synthesiser is using MIDI to control a second or third, then there is no longer any necessity to have more keyboards than the one on the master instrument. As MIDI use becomes more general, so more 'synthesiser modules' are appearing on the market. These are simply the sound-generating and sequencing units that were formerly part of keyboard instruments. As these modules have little or no *visual* interest, there is no need for them to be present on stage.

There is another development that pre-dates MIDI, but which is likely to gain more attention as the number of keyboards diminish. This is the possibility of having electronic sound synthesis controlled by string-playing data from guitars, and by breath and mouth control data from wind instruments. Compared to the mechanical action of simply depressing a note on a keyboard, plucking a string or vibrating a reed involves more complex acoustic information being transmitted to the instrument in question. If this information is digitally encoded and transmitted via MIDI to an

off-stage synthesiser module, there is no reason why a saxophone player cannot be the primary controller of a group's electronics — even of its drum machine. Yamaha has incorporated breath control into its DX7 synthesiser, and the SynthAxe — a recently developed digital guitar — has been designed to use MIDI for controlling the output of a Fairlight.

This means that a string sound on the Yamaha could be produced with the unique performance characteristics of a saxophone, and, similarly, a Fairlight trombone sample could be articulated by strumming a guitar. Although neither of these developments is imminent — after all, the SynthAxe is a very expensive 'guitar' at £7,000 — they indicate probable tendencies for live performance in the near future. There is likely to be a gradual reduction in on-stage keyboards; string, wind and possibly tuned percussion instruments such as vibraphones will become more important, and, as sound-sampling technology becomes cheaper, acoustic sound may well be predominant.

By the end of the 1980s, musical traditionalists may be relieved to see — once again — a jazz-style unit of guitar, saxophone, double bass and drums. They may, however, be confused by the fact that the guitarist is playing an invisible vibraphone and the sax-player is 'breath-drumming'.

For many groups used to live performance, the first experience of an advanced recording studio can be intimidating. They are presented with musical instruments and operating systems they have never encountered before and are allocated a producer who may not know their work or their intentions particularly well. Yet they are expected by their record company to produce 'bigger and better' versions of their stage hits in this unfamiliar environment. It is comparable to putting a semi-professional theatre company onto a big-budget film set and expecting an instant box-office success. Sometimes the transition is successful, and all concerned are satisfied. But very often, the original ideas get lost in a maze of studio devices, and the group is left with an expensive failure.

Most often the breakdown point occurs when the group discards its own familiar equipment — and, in effect, its own 'sound' — and takes to the more desirable instruments available in the studio. But an idea which worked on a Mini-Moog synthesiser may fall apart when played on a Fairlight digital sampling computer and, if this type of musical failure occurs often enough, then the justification for using such a studio begins to weaken.

If, however, the musicians in the group are familiar with MIDI, and if a microcomputer has been used for storing sequences and other musical control data, then they will be familiar with the procedures already in use in advanced studios. On the most immediate level, it should be possible to try out ideas using a succession of different studio instruments with the minimum of difficulty, and with the prior knowledge that ideas and sequences can be transformed simply by swapping synthesisers.

A MIDI background is also invaluable when getting to grips with studio systems other than those directly involved in sound-generation. A solid-state logic mixing desk, for example, has a dedicated computer that will recall and re-run any series of decisions made in the final stages of recording — known as mixing. When all the music has been recorded onto 24 separate tracks of tape — guitar on one track, backing vocals on another, lead vocals acrosss three others, and so on — the crucial task of balancing and mixing all the elements begins. It is usually at this point that individual parts are treated with any required 'effect' to enable them to stand out or blend in the mix. A single trumpet note may need reverberation added at one point only, and with 23 other things happening at the same time it is easy to miss it. Using a computer to handle such incidents while mixing is like MIDI-sequencing on a grand scale.

Another technique, originally developed for video synchronisation and editing, but emerging more into music production, is the use of time-code. Time-code is like a digital clock and trigger signal, but is laid down onto tape. It uses 80-bit words to provide synchronisation data when recording music against video sequences, and enables musical events and split-second video edits to be sequenced together.

Musicians, therefore, have good practical reasons for acquiring MIDI-compatible instruments — but, in addition, MIDI is a good introduction to the more advanced music systems currently in use. In the next, and final, instalment, we will take a look at some of the more exotic computerised music equipment in use today.

Big Science
Laurie Anderson, New York poet and performance artist, combines an unusual mixture of sounds and sound equipment with film, tape, and video technology to create a unique style. In songs like 'O Superman' and 'Mr Heartbreak', she uses or is backed up by anything from African bells to state-of-the-art electronic instruments such as the Vocoder and Synclavier

PIECE WORK

We have already seen that the LOGO user can define procedures to carry out sequences of commands. Procedures, once defined, may be used in exactly the same way as LOGO 'primitives' (the basic commands of the language). It follows, therefore, that we can use procedures in the definition of further procedures. We show you how.

As an illustration of this principle, let's consider the tangram puzzle. This is a square that has been divided up into seven geometric pieces, which are combined in various ways to form different shapes. In our example, we will use the seven basic pieces to create a shape that resembles a dog. We start by defining LOGO procedures for each basic piece; these 'piece procedures' are then incorporated into a further procedure, which is given the name DOG. As the turtle must be correctly positioned before each piece is drawn, other procedures — MOVE1 to MOVE7 — must also be used.

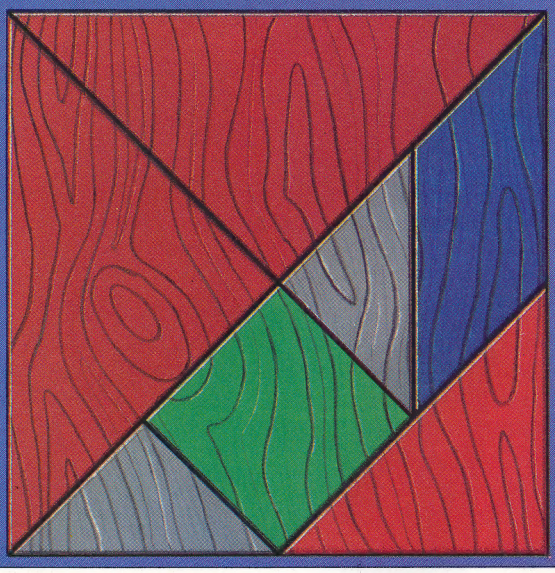
It would be just as easy to produce this drawing by simply stringing one command after the other in one long procedure. Our method uses the principles of 'top-down' design. We have covered this subject in some detail (see page 476), but, very roughly, it simply means breaking a problem up into a number of parts and then proceeding to solve each part in turn. The great advantage of this approach is that the LOGO programmer may define a procedure containing subprocedures that have yet to be defined. The main procedure cannot be

Shaping Up

The tangram puzzle is a collection of seven shapes that can be mixed in various combinations to create simple designs. Here we list LOGO procedures to draw the seven basic tangram pieces, as well as a sample program that draws the figure of a dog. The DOG procedure begins drawing with the triangle for the dog's hind leg

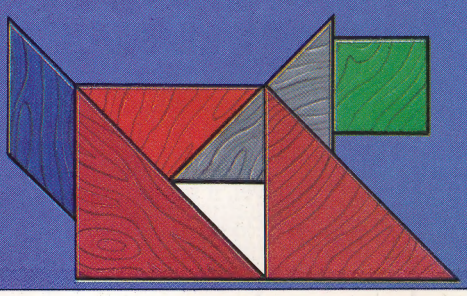
Tangram Procedures

```
TO SQUARE
   REPEAT 4 [FD 25 RT 90]
END
TO PAR
   REPEAT 2 [FD 25 RT 45 FD 35 RT 135]
END
TO TRI1
   FD 25 RT 135 FD 35 RT 135 FD 25 RT 90
END
(There are two triangles of this size)
TO TRI2
   FD 35 RT 135 FD 50 RT 135 FD 35 RT 90
END
TO TRI3
   FD 50 RT 135 FD 71 RT 135 FD 50 RT 90
END
(There are two triangles of this size as well)
```



Dog Program

```
TO DOG
   TR13 MOVE1 PAR MOVE2 TRI2 MOVE3
   TRI1 MOVE4 TRI3 MOVE5 TRI1 MOVE 6
   SQUARE MOVE7
END
TO MOVE1
   PU FD 15 LT 45 PD
END
TO MOVE2
   PU RT 45 FD 35 LT 45 BK 35 PD
END
TO MOVE3
   PU LT 45 BK 25 PD
END
TO MOVE4
   PU RT 90 BK 25 PD
END
TO MOVE5
   PU FD 50 RT 45 PD
END
TO MOVE6
   PU FD 25 RT 135 FD 5 LT 90 PD
END
TO MOVE7
   PU LT 90 FD 5 RT 45 BK 25 RT 45
   BK 50 LT 90 BK 50 PD
END
```



KEVIN JONES

run, of course, until the subprocedures have been written or dummy routines provided in their place. To show how this works, let's consider how the dog-drawing program was constructed.

The DOG procedure was written first, even though none of its component procedures yet existed. We then wrote each of the shape-drawing procedures separately. These were followed by the positioning procedures. Each time a new procedure was written, DOG was run to ensure that everything fitted together properly. When LOGO came to a MOVE procedure that had not been written it stopped with an error message. However, it was easy to tell from the drawing whether everything up to this point was correct, or if there was an error in the last MOVE procedure.

Our set of procedures demonstrates another important point — each of the shape procedures, and the DOG procedure itself, leaves the turtle state unaltered. That is, the turtle is at the same position with the same heading at the end of the procedure as it was before the procedure was run. Such procedures are said to be *state transparent*. Making procedures state transparent eases the task of putting procedures together to construct more complex drawings. Take the DOG procedure, for example: once the turtle is positioned we know that after drawing a piece the turtle will return to the position it was in when it started that piece. So we need know nothing about the internal workings of the procedures in order to put the pieces together. By making DOG state transparent, we make it easier to use this procedure as part of another — for example, we could draw a whole screenful of dogs.

LOGO WORKSPACE

By now you will have a fair number of procedures in the computer's memory — so let's take a closer look at LOGO memory organisation. LOGO's working memory consists of a list of *nodes* (each of five bytes). Once LOGO is loaded, you will have between 1,000 and 3,000 of these, depending on the machine you use. As procedures are defined, these nodes are used up. Other nodes may be used as procedures are run or if variables (to be discussed later in the course) are used.

The procedures you have defined constitute your *workspace*. You can see which procedures are held in the workspace by entering POTS (for PRINTOUT TITLES). To look at an individual procedure, use PO (for PRINTOUT) — for example, PO SQUARE. If a procedure is no longer required, workspace can be freed by using ERASE — the command ERASE SQUARE would remove the procedure called SQUARE from memory. Erasing a procedure releases the nodes used. LOGO will mark these nodes, but will not yet add them to the list of free nodes; instead it will continue working with its present free nodes list until all of these have been used up. It will then go through its memory, gathering up all the nodes that have been released and using these to form a new list of free nodes. This process is referred to as *garbage collection* and is the reason why LOGO seems to hesitate for a second or two from time to time.

SAVING PROCEDURES

In order to make permanent records of your procedures on disk, you must save the workspace as a file. Using MYPROCS as an example file name, you would type SAVE "MYPROCS (note the quotation marks before, but not after, the file name). The workspace itself will not be affected by this. The file may be loaded with READ "MYPROCS. This causes the procedures in the file to be defined and added to the current workspace. If a procedure is defined with the same name as a procedure already held in the workspace, then the new definition replaces the earlier one.

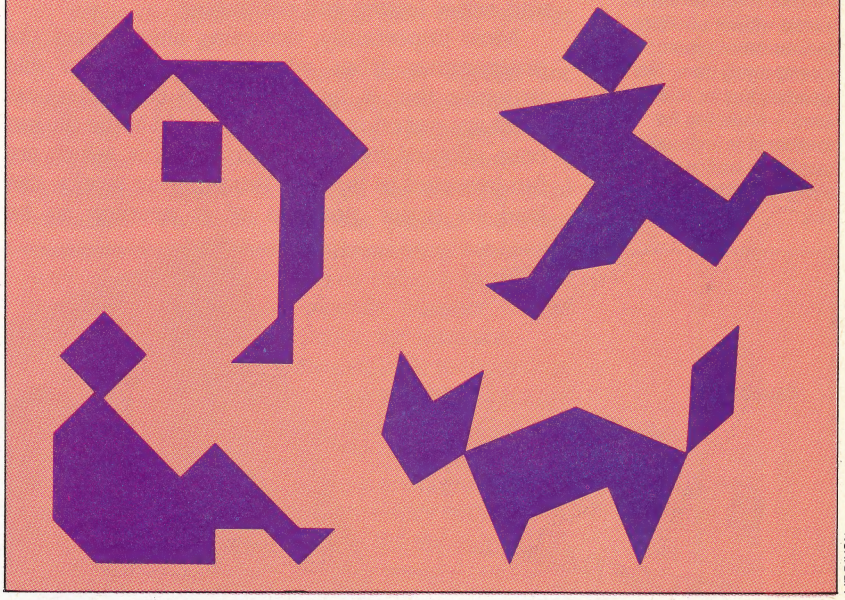
Other useful disk-handling commands are CATALOG and ERASEFILE. CATALOG gives a list of all the files on the disk, and ERASEFILE "MYPROCS would erase the MYPROCS file from the disk. Cassette-based versions of LOGO use different commands — the relevant manual should be consulted for these.

Abbreviations

ERASE	ER
PRINTOUT	PO
PRINTOUT TITLES	POTS

Procedure Problems

1) Write procedures for the other tangram shapes shown. (You will first have to solve the puzzle of how to construct the shape from the different pieces, of course!)
2) Write a procedure to draw a 'house' (simply an equilateral triangle above a square).
3) Write a procedure to draw a five-by-five board of squares.
4) Rewrite the procedure used earlier to draw a six-pointed star so that it uses subprocedures.



LIZ DIXON

Logo Flavours

In all LCSI versions, procedure names must be prefixed by a quotation mark if they are inputs to PO or ERASE: for example, PO "SQUARE and ERASE "SQUARE.

To read a disk file use LOAD "MYPROCS.

Versions supporting cassette tapes or microdrives have somewhat different commands from those for disks. You should consult the relevant technical manual.

LENDING LIBRARY

It is extremely useful to develop techniques that make more efficient use of the time and effort spent in programming. We discuss one such method — creating libraries of routines that can be merged into programs — and list the sort of details that must be taken into account when programmers share the task of coding.

Following the structured design methods that we have already described in this course may seem like a long-winded approach — but it does, in fact, save time (not only in the coding but especially in the debugging of a program). This is because programs that are created at the keyboard tend to have unnecessarily complicated structures and algorithms, which means that they take longer to write, are more prone to error and, because they are more difficult to follow, take much more effort to test and debug. Planning the program in advance simplifies the structure and the algorithms and thus leads to fewer coding errors and easier testing and debugging.

Most importantly, designing ahead saves the programmer from writing a control or file structure that is later found to be inadequate (perhaps not enough space in a field in the file has been allowed for). Problems like this, which are fundamental to the way the program works, can lead to major portions of it needing to be rewritten.

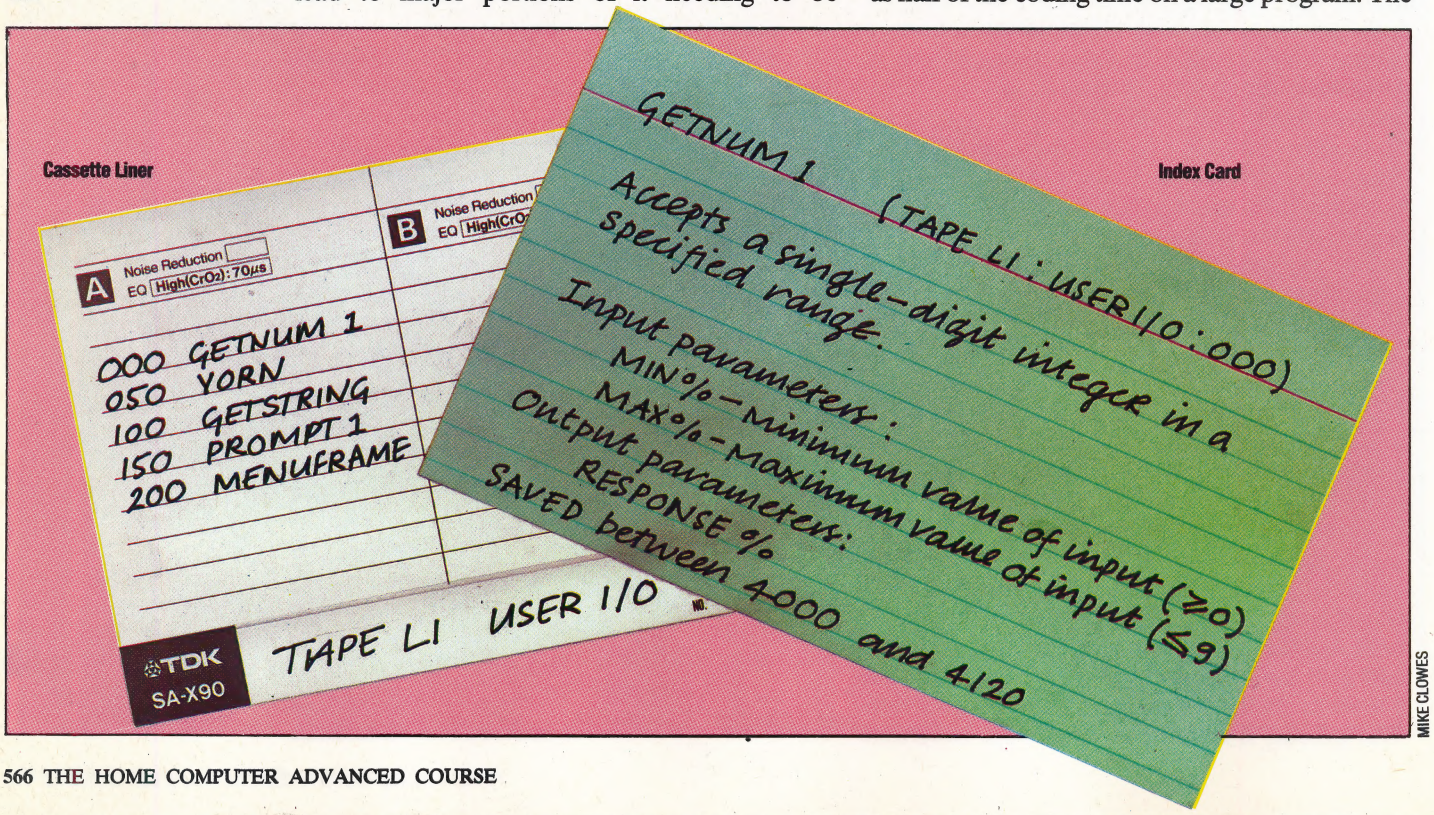
Those with a 'proper' typewriter-style keyboard may like to invest some time in learning to touch-type. Apart from this, though, there is little that may be done to increase the speed at which program lines are entered at the keyboard. However, the process of coding programs may be greatly speeded up in several ways. The first is the simplest: invent, adopt and use a number of 'conventions' when coding. Such measures include: using particular types of name for local variables to differentiate them from main program variables; beginning each subroutine at lines ending in 000; ending each subroutine with RETURN on a line of its own; starting each type of subroutine in a particular block of lines (file-handling routines between 9000 and 9999, utilities at 50000 onwards, and so on).

The benefits of using these conventions are numerous: you don't have to hunt for the menu routines because you know that they are always in the same place; you don't have to worry about whether you have used the same variable name in the main program and in a subroutine — because its name will indicate it is a local variable.

PROGRAM LIBRARIES

Such coding techniques are also useful when libraries of programs are created. A well-organised library of subroutines can save as much as half of the coding time on a large program. The

Uniform System
Libraries of subroutines are useless without a uniform documentation system accompanying them. This is especially true for cassette users — inspecting the contents of an undocumented cassette by loading and listing each program is a thankless task



best way to start such a library is to go through existing programs and take out all the subroutines that are well written and have some general applicability (I/O routines, date routines, upper to lower case conversion, and so on). Each routine should be saved as a separate file, and these should be grouped together according to function (if they are to be stored on tape then each function group should be stored on a separate cassette) with meaningful file names to identify them. Keep a card index or a database of the file names, together with a description of what each routine does.

Needless to say, it is important to ensure that all library routines are thoroughly tested and debugged. They will be used in programs for which they were not specifically designed, so make sure that they will trap any illegal input values. You should also ensure that any values output from the library routines will not cause problems to the program that uses them. Make each routine as efficient as possible and include as much internal documentation as is necessary for you to understand the routine's function at a later date. Add to the collection as the need arises — there is no point in adding new routines 'on spec' as experience shows that this is largely wasted effort. Don't forget to number the lines of the library routines according to the convention established (this will save on RENUMbering when the routines are merged into a new program). Useful library routines may be found in computer magazines, which often publish routine listings as well as complete programs (and these can be cannibalised to obtain the useful subroutines).

To make use of a library like this, it is necessary to have a way of merging routines together to form a complete program. For those using compiled languages, a 'link-loader' or similar program is usually supplied; this takes compiled modules and joins them to make an executable program. For BASIC programmers, unless a compiler is available, the easiest way to achieve this is to use a combination of RENUMber and MERGE commands. To merge a library routine into the new program, first load the program, decide where the library routine will go and make sure there is a large enough block of unused line numbers for it to fit in. If necessary, RENUMber the library routine so that it will go into the space allotted to it. Then use the MERGE command to join the two programs; check that everything works as it should and SAVE the new program with the library routine in place.

GROUP EFFORTS

It is often the case that home computer users work together in groups to write programs — either at school or in their user clubs. Most of what has been said about program design and programmer efficiency is particularly relevant to such team efforts. In fact, most of these ideas and the concept of structured programming were developed in order to split the workload of commercial programming projects. Thus, a number of different programmers could work on different parts of the same program at the same time to produce a working program.

For BASIC programmers to work like this, it is essential to agree on the conventions to be used when coding. Assuming that a design has been agreed on, the programmer of an individual module needs to know:

1) What the files will be and how they will be organised.
2) What conventions have been agreed for naming variables. The most important variables, such as arrays that are used throughout the program, should be named in advance. A convention should be agreed for naming local variables. Variables that are passed between modules should either be named in advance or a way of ensuring that each is unique should be devised — adding the module number of the originating module as a suffix, for instance.
3) What library routines are available to the group, the format of each of these, how their variables are named, what they do, and how well tested and debugged they are.
4) How error-handling routines are organised (for instance, whether each routine copes with its own errors or whether the routines set an error 'flag', which is then dealt with by the control routine).
5) The exact function of any module that is being written.
6) The exact range and type of data that each individual module will accept as input and return as output.

This implies a lengthy planning stage with many meetings to agree strategy, followed by a short programming stage. Testing — including the testing of group-produced programs — will be dealt with later in the course. The next instalment will concentrate on the design of programs that will run faster and use less memory.

MERGE

- The Spectrum has the straightforward version of the command: it merges the named file with the program in memory; the incoming line overwrites the existing line in the event of a line number collision.
- With the BBC Micro *SPOOL command you can create ASCII versions of the program files, then write a BASIC program (or use a word processor) to access these files, one program line at a time. Merge the two files into a third ASCII file, and convert it into a program using the *EXEC command.
- On the Commodore: OPEN 1,1:CMD1:LIST:PRINT #1:CLOSE 1 creates on tape an un-named ASCII file of the program in memory. LOAD the other program, and add to it a routine to INPUT and print the ASCII file lines on screen. Stop the program and RETURN over the screen, thus merging the two programs

FIRM OFFER

With the introduction of the Macintosh, Apple Computer has firmly established its name in the UK market. More recently, the company has turned its attention to upgrading its existing line of 6502 machines — the Apple II family. We examine the new Apple II*c* computer, and consider the company's marketing strategy.

The success of the Macintosh, and increased competition — in the US home computer market from Commodore and in the world business market from companies like ACT and IBM — had put the future of the Apple II range of computers in some doubt. Many dealers and industry analysts predicted that the range was nearing the end of its market life, despite Apple's insistence that it would remain committed to the 6502 machine and its large user base. To demonstrate its support, the company recently launched the Apple II*c*, as well as software and hardware upgrades for the older Apple II lines. The new products are expected to extend the market life of the Apple II by as much as three years.

In its various forms (II, II+, and II*e*), the Apple II helped to create the personal computer market, dominated US computer sales for several years and contributed to Apple's total sales record of over $1 billion. There are over two million Apple computers in use worldwide, yet the Apple II has never reached the same level of sales success in the UK or other parts of Europe, primarily because of ineffective pricing and marketing policies. At £1,500 (including monitor and disk drive) the machine was priced much too high to be considered a home computer. And interference from Apple headquarters in California is often cited as the reason why Apple has never gained the kind of share of the UK educational or business markets that it has in America. Nevertheless, the relatively small group of Apple users in the UK tends to be fiercely loyal to the machine.

The latest incarnation of the Apple II is the II*c* (the *c* stands for compact). It is smaller than its predecessors by about half, yet houses a half-height 5¼in disk drive in the side of its casing. At 3.4kg (7.5lb), the II*c* is meant to be transportable, and is clearly designed to be used during the day at work and then to be carried home at night. Towards this end, the II*c* has a small carrying handle moulded into its plastic case, and a choice of connectors (for use with a composite or RGB monitor at work and a standard television set at home). The carrying handle folds back to prop the machine up into a comfortable working angle. This also keeps air circulating around the machine to prevent overheating.

Unlike the previous Apple II models, the II*c* is a closed system, with no expansion slots inside. Instead, Apple has built several of the most important options into the machine. These include the monitor and television display ports; a joystick port that also supports the optional mouse; a modem port; a printer port; an audio output socket and a connector for a second disk drive. The interfaces are labelled with icons — small pictorial representations of their function. The II*c* also has a built-in 80-column display, and 128 Kbytes of RAM. Most of these features are optional on the II*e*, and would require the addition of at least three plug-in expansion cards.

The Apple II*c* has a 63-key QWERTY keyboard, with a similar layout to the II*e*. The Reset key, however, has been moved to a position above the left edge of the keyboard, and two small switches have been added next to it. The left switch toggles the screen display between 40 and 80 columns. The owner's manual recommends that you use a 40-column display when working with a television set, and an 80-column display for a monitor. (Some of the existing Apple software will

Portable Upgrade
Apple's new, upgraded portable version of the Apple II, is the II*c*. The II*c* has 128K of RAM, an 80-column display, a variety of interfaces and a built-in disk drive. The II*c* costs £925, and is shown here with the optional green phosphor monitor

CHRIS STEVENS

display only 40 columns, regardless of the switch's position.) The second switch toggles between the European character set and the North American characters shown on the keyboard. This is useful when a character is required that can only be found in one of the two sets (such as '#', which is replaced by '£' on the European keyboard). There are two lights above the right edge of the keyboard: one indicates when the power is on, and the other lights when the disk drive is in use.

When you power up the IIc, the disk drive automatically starts spinning and looks for a disk. It will continue to spin until a disk is found, or until the Reset key is pressed while the Control key is held down. With no disk in place, the IIc loads Applesoft BASIC from ROM. Applesoft has been virtually unchanged since the Apple II+ was introduced. It varies only slightly from early versions of standard Microsoft BASIC. The use of Applesoft makes it possible for the IIc to run Applesoft programs written for earlier models. Unfortunately, Applesoft lacks many of the programming structures available in more advanced dialects, such as BBC BASIC. For example, Applesoft has no RANDOMIZE feature, AUTO line numbering facility, IF . . . THEN . . . ELSE structure or WHILE command. It also lacks CIRCLE and PAINT commands for graphics programming.

When there is a disk in the disk drive, the IIc runs either DOS 3.3 (Apple II+ and IIe disks) or the new Apple operating system, PRODOS. This is a derivative of the operating system Apple designed for its first business system, the Apple III. PRODOS has a hierarchical (tree) filing structure. Disk files are stored in much the same way that documents are stored in a filing cabinet. Thus, all files that relate, for example, to project ZED could be filed on the disk under the heading ZED. The accounting files of project ZED — such as Costs, Sales, Revenues — could be gathered into a group called Accounts. With a manual filing system of this type, when you wanted to find the file for Sales, you would first open the main file, ZED, then open the Accounts file, and finally pull out the file marked Sales. In PRODOS, this is done via a 'pathname' that lists the appropriate file names in order, so the process we have just described would be accomplished by typing the file names, separated by slashes:

```
/ZED/ACCOUNTS/SALES/
```

Pathnames can be up to 64 characters long. This process may seem complicated, and in fact it does take a while to get used to, but in the long run it simplifies the organising and management of disk files. Tree filing systems like PRODOS are also used in MS/DOS, the operating system used on the IBM PC.

The screen display of the IIc is also an improvement on previous models. Besides two text options (24 lines by 40 or 80 characters), the IIc has three graphics screens: 40 by 40 (Lo-Res), 280 by 192 (Hi-Res), and 560 by 192 (called Double Hi-Res). There are 16 colours available.

Cautionary Tale

CHRIS STEVENS

Apple III

The Lisa

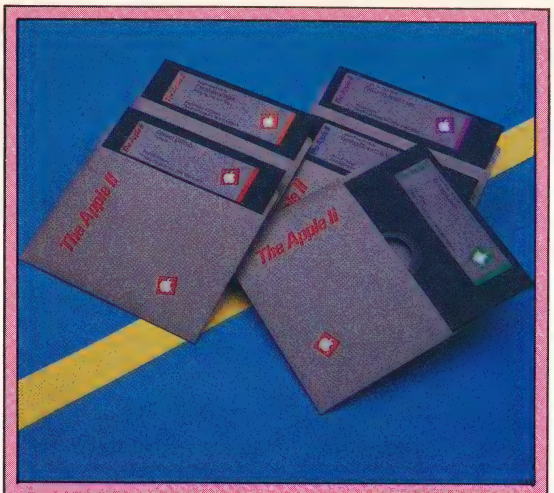
Realising the importance of the business computer market, Apple developed the Apple III, a brilliant desktop machine that makes use of two 6502 processors to support up to 512K of memory. The Apple III runs an operating system called SOS (Sophisticated Operating System), that permits the III to communicate with a hard disk storage unit and stores files in a hierarchical file structure. This tree structure formed the conceptual basis for the Lisa operating system, and many of its features appeared in MS/DOS on the IBM PC. Unfortunately, the III had a number of hardware difficulties immediately after it was introduced, and earned a bad reputation. Apple recalled and replaced all the faulty machines, but the III never overcame the bad publicity it received, nor the feeling of many users that the operating system was too complicated. As a result, the Apple III is no longer on sale.

The Lisa, introduced in spring 1983, was the first mass-market personal computer to use integrated, windowing software, had an operating system based on pictures rather than words, and was operated by a hand-control device: the mouse. The Lisa created so much excitement that it caused Apple company stock to jump from $24 per share to over $60 in the space of a few months. Developed at a cost of over $50 million, the Lisa was priced too high and aimed at the wrong market. Apple originally targeted the Lisa for executives of large corporations. While some senior executives have been very impressed by the Lisa, most of its sales have gone to small companies in specialised businesses, like advertising, graphic design, and public relations. The Lisa did not sell as well as expected, and Apple's stock plummeted to $17 a share, again in the space of a few months. The Lisa was recently replaced by the less expensive Lisa II. Under the guidance of John Scully, imported from the Pepsi Cola Company to guide Apple's operations, the innovative computer company appears to have learned from its mistakes. The Macintosh is selling in huge numbers around the world, recovering some of the Lisa/Mac development costs, and the IIc seems poised to breathe renewed life into Apple's breadwinner, the Apple II

Apple offers a green phosphor monitor for £140, and is expected to introduce a flat-panel LCD display soon. The LCD screen will be manufactured for Apple by Sharp, and will have a full 24-line by 80-character display. It will probably cost about £500. It is expected that a battery pack for the system will be available soon, and this will enable the Apple II*c* to become a fully portable computer.

The greatest advantage the II*c* has is its software base. Over 17,000 programs have been written for the Apple II. Although some of these programs are available only in America, you can still be fairly certain that anything you could want to do with an Apple has probably been done already, and appropriate software has been written. The software base includes some of the world's best games programs (CHESS 7.0, ZORK, Microsoft Flight Simulator, Pinball Construction Set); a wide variety of word processors, spreadsheets and database programs; accounting programs; graphics design programs; scientific laboratory control programs and educational programs (from beginners' readers to advanced calculus).

In addition to existing II and II*e* software, Apple has introduced a program called Appleworks, an integrated word processor, spreadsheet and database with windowing. Appleworks is quite sophisticated and easy to use. The II*c* comes with a disk that introduces you to Appleworks, although it is not a working copy of the program. As is typical with Apple, the company presumes that you will eventually want to buy this £175 package. Other disks provided with the system are a similar introduction to Apple LOGO; Apple Presents Apple, an interactive introduction to the basic system; a very simple introduction to BASIC programming, and the PRODOS systems utilities disk. MousePaint, a mouse-driven drawing program based on MacPaint, is also available. MousePaint is supplied with the Apple II mouse and costs £70.

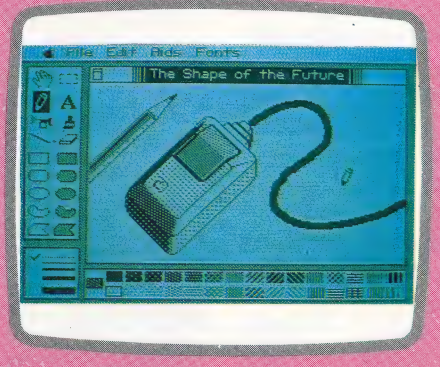


Apple Disk Pack
The Apple II*c* comes with a disk pack containing five disks. Four of them introduce the workings of the machine, BASIC programming, and optional applications programs that Apple hopes you will buy. The fifth is the system operations disk with PRODOS, Apple's new disk filing system for the Apple II line

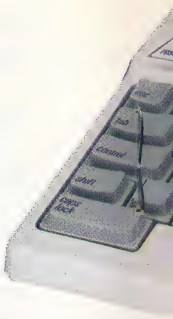
The II*c* comes with a small pamphlet that describes how to set up the system, and a 142-page owner's guide that briefly and clearly explains system operations and the use of the five disks that come with the computer. The manuals are well-written and colourfully illustrated. They are obviously geared toward the first-time user.

The II*c*'s design is very attractive and stylish. Apple dropped the beige plastic used on the II, the Macintosh and the Lisa in favour of a bright white finish. The 'racing stripe' lines across the top of the case let air flow through to the circuits to keep the system cool.

The Apple II*c*, like its predecessors, is a great desktop machine for the office. With the added LCD screen and battery pack, which are expected soon, it should earn a reputation as a useful, portable workmate. Had the price been right, it could also have been a popular home computer.



MousePaint
The Apple II mouse is available for the II+, II*e* and II*c* models for £70, and comes with MousePaint. MousePaint is based on MacPaint, but it is a scaled-down version of the Macintosh program. It allows you to use the mouse to create pictures very easily, and is written to take advantage of the II*c*'s high resolution screen display. To use the mouse with other Apple II computers, you must have an extra interface card that plugs into one of the internal expansion slots



Composite Video Output
RS232 Printer Port

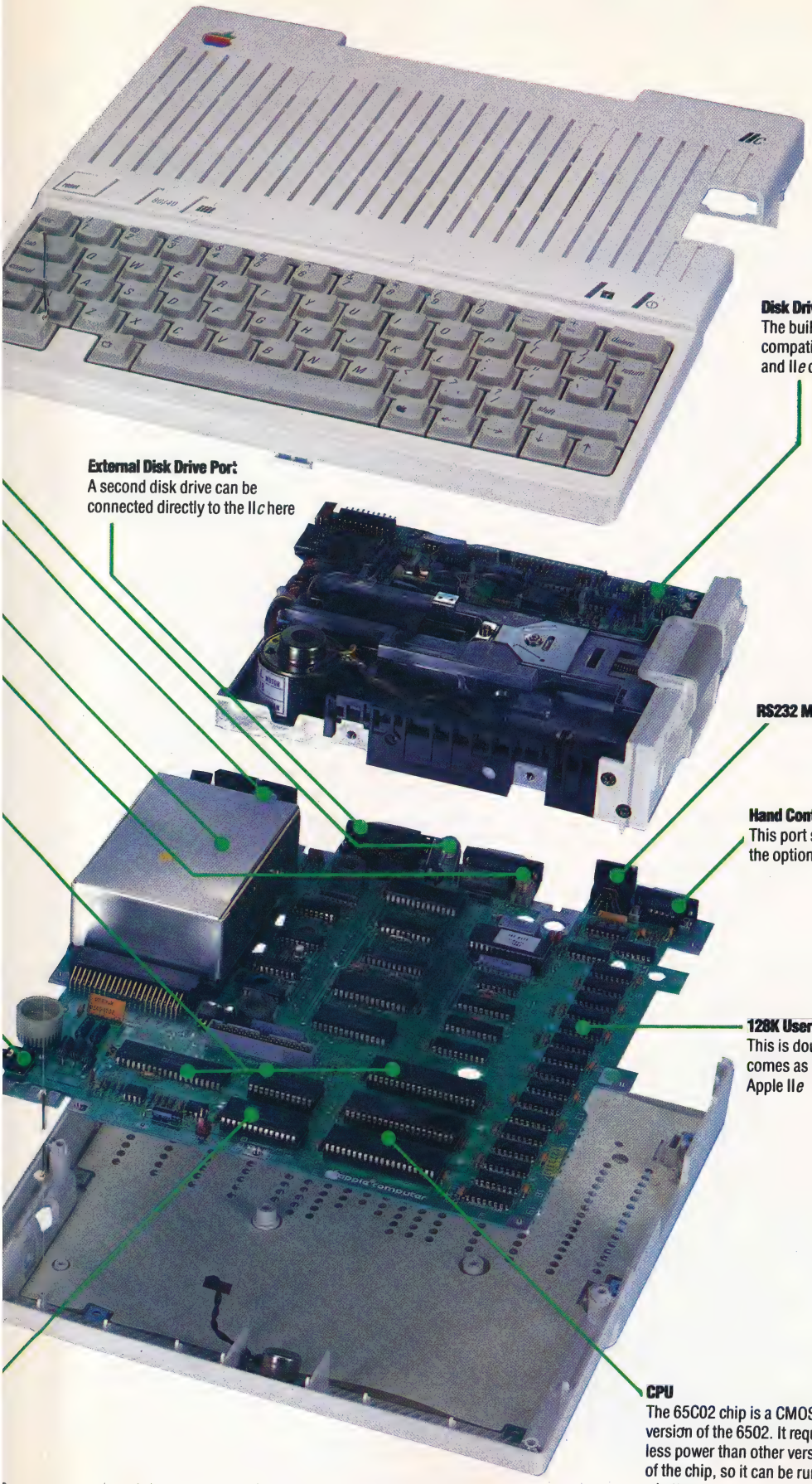
Power Supply
The II*c* has an internal 12-volt power supply, but still requires a transformer box for 240-volt mains input

RGB Video Output
By plugging a small PAL adaptor into this port, the II*c* can connect to a standard television set

Input/Output Controllers
These chips control keyboard operations, and input and output ports

Audio Socket
This can connect the II*c* to an external hi-fi amplifier

ROM
This holds Applesoft BASIC and the housekeeping routines



Disk Drive
The built-in 143K disk drive is compatible with most Apple II+ and IIe disks

External Disk Drive Port
A second disk drive can be connected directly to the IIc here

RS232 Modem Port

Hand Control
This port supports a joystick or the optional mouse

128K User RAM
This is double the amount that comes as standard with the Apple IIe

CPU
The 65C02 chip is a CMOS version of the 6502. It requires less power than other versions of the chip, so it can be run from a battery

APPLE IIc

PRICE
£925

DIMENSIONS
305 x 292 x 64 mm

CPU
6502 @ 1MHz

MEMORY
128K RAM, 16K ROM

SCREEN
24 lines of either 40 or 80 characters. Three graphics display modes with maximum resolution of 560 x 192 pixels and 16 colours.

INTERFACES
9-pin joystick port that doubles as a mouse port; RS232 modem port; RGB (or PAL TV) output; composite video output; external disk drive port, and RS232 printer port

LANGUAGES AVAILABLE
Applesoft BASIC resident in ROM; LOGO, PASCAL, FORTRAN

KEYBOARD
63-key typewriter-style keyboard with four cursor keys, international character sets

DOCUMENTATION
A very colourful and easy-to-understand owner's guide comes with the machine. The guide is clearly designed for the first-time user. There are also two slim manuals to help you set up the IIc and work with the systems utilities disk

STRENGTHS
The IIc's greatest strength comes from its compatibility with the Apple II, which means that it can run most of the more than 17,000 titles available for the other Apple

WEAKNESSES
The Applesoft BASIC, which hasn't changed noticeably in six years and is beginning to show its age, lacks flexibility. The IIc's price may put it out of the range of many home users.

TYPE CAST

The version of BASIC used on the Commodore 64 is not without limitations, although these are seldom a great obstacle to thoughtful and creative programming. We give you a program that enables you to define your own character set.

The Commodore 64 is capable of producing splendid sound and graphics — as much commercial software amply demonstrates — but its BASIC does not support a single 'purpose-built' colour or sound command. The BEEP, DRAW, INK and PAPER commands supported by Spectrum BASIC, for example, have no equivalents among the sparse set of commands available to the Commodore 64 programmer. The result is that most BASIC programs have crude sound and graphics, and even the best programs tend to contain many DATA and POKE statements — as the listing with this article shows. The character-generating program that we list here makes the process of defining new characters less demanding by allowing you to design them on the screen (rather than POKEing values straight into RAM); these definitions are then automatically POKEd into memory.

We have already investigated in some detail the procedure involved in defining your own characters on the Commodore 64 (see page 232). The essential preliminary actions are performed in the subroutine at line 61000. The top of user memory is lowered from location 40959 to 14335. The entire two Kbyte upper case character set that the Commodore has resident in ROM (from address 53248 onwards) is then copied into RAM (14336 onwards) where it can be accessed and manipulated using PEEK and POKE statements. Finally, the VIC (video interface chip) is switched to address the relocated character set.

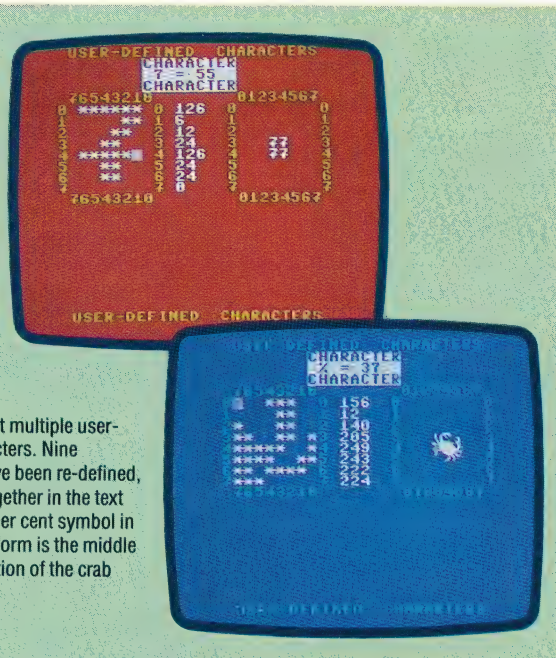
Once the character set has been relocated in RAM, two 'windows' are displayed on the screen by the initialisation routine, and control passes to the input routine at line 2500. This routine scans the keyboard and maintains a flashing cursor in the left-hand or 'edit' window. The character currently being redefined is displayed (suitably magnified) in this window, with the values of its eight defining bytes next to it.

The unshifted function keys (f1, f3, f5, f7) control cursor movement inside this window. The cell under the cursor (corresponding to a bit in one of the eight definition bytes) can be toggled on or off with the shifted function key f2. When this happens, the eight definition values are updated, and all occurrences of the character elsewhere on the screen can be seen to change immediately.

Pressing the shifted function key f4 allows you to replace the character in the edit window with another character. Characters are described by their POKE (or screen code) values as listed in Appendix F of the User Manual. These values are not the same as the CHR$ codes (although there is a correspondence), but are more convenient to use here since the character definitions are arranged in memory in the order of these codes.

The shifted function key f6 allows you to write a 'character-sized' copy of the character being edited into the right-hand (or 'text') window, at the position corresponding to that of the edit cursor. If the cursor is in the top left corner of the edit window, for example, and 'A' is the character being edited, then an 'A' will be written in the top left corner of the text window when f6 is pressed.

Finally, pressing the exclamation mark key STOPs the program; CONT will restart it. When you 'quit' or exit from the program, you can type NEW and then LOAD another program without disturbing your re-defined character set. However, there are problems involved with this. First of all, the top of user memory has been lowered so that there are only 12 Kbytes available for the new program. Secondly, switching off the machine destroys the new set. Both problems will be discussed in a future article. In the meantime you should note the definitions of your newly-defined characters and use them, if necessary, as shown in the sample program on page 233.



Sans Serif
In the left-hand, or edit, window is the re-defined version of character number 55 — the figure seven. The serif has been erased, and the continental cross-bar added. In the right-hand, or text, window are four copies of the seven. The changes are visible in these sevens and in those around the windows, but the seven in the status box retains its original definition

Claws
Sprites are just multiple user-defined characters. Nine characters have been re-defined, and placed together in the text window. The per cent symbol in its re-defined form is the middle rightmost section of the crab

IAN McKINNELL

Defining Characters

```
19 REM***********C64***************
20 REM*   USER-DEFINED CHAR-GEN   *
21 REM***********C64***************
50 POKE 52,56:POKE 56,56:CLR
60 PRINT CHR$(147)"PLEASE WAIT 22
     SEC'S"
70 GOSUB 61000:  REM COPY CHAR.SET.
100 GOSUB 1000:  REM INITIALISE
120 FOR CT=0 TO 1 STEP 0
140 GOSUB 2500:  REM INPUT
160 GOSUB 3000:  REM VALIDATE
180 ON PT GOSUB 3500,4000,4500,7000
200 NEXT CT
900 END
999 REMM*********************
1000 REM*  INITIALISE        *
1001 REM*********************
1020 DIM BD(8,8),C$(2,2),OF(2,7),
     CX(4)
1040 SH$=CHR$(19):SC$=CHR$(147):R$=
CHR$(18):N$=CHR$(146):CD$=CHR$(17)
1060 P$=CD$+CD$+CD$:P$=P$+P$+P$+P$:
     P$=P$+P$:P$=SH$+P$
1080 C1$=CHR$(144):C2$=CHR$(5)
1200 REM-----INITIALISE SCREEN-----
1210 SO=1024:PRINT SC$C1$
1220 RO=4:CO=3:RL=8:CL=8:OF=16
1230 Z$="USER-DEFINED  CHARACTERS"
1240 C=4:GOSUB2100:R=24:GOSUB2100
1250 L$=" 76543210 ":S$="        "
1260 Z$=L$:R=RO:C=CO:GOSUB 2100
1270 C=CO-1:FOR R=RO+1 TO RO+8
1280 Z$=STR$(R-RO-1):Z$=Z$+S$+Z$
1290 GOSUB2100:C=C+OF:GOSUB2100
1300 C=C-OF:NEXT R
1310 C=CO:Z$=L$:GOSUB2100
1320 L$=" 01234567 "
1330 Z$=L$:R=RO:C=CO+OF:GOSUB 2100
1350 C=CO+OF:R=RO+CL+1 :GOSUB2100
1370 PRINT C2$
1400 C$(1,1)=R$+" "+N$:C$(1,2)=" "
1410 C$(2,1)=R$+"*"+N$:C$(2,2)="*"
1420 REM-----CURSOR    OFFSETS-----
1440 DATA 0,-1,+1,0,-1,0,0,+1
1460 FOR K=1 TO 2:FOR L=1 TO 4:
1480 READ OF(K,L):NEXT L,K
1500 REM-----CHAR  CONVERSIONS-----
1520 DATA 64,0,32,64
1540 FOR K=0 TO 3:READ CX(K):NEXT K
1620 RP=1:CP=1:CN=1:GOSUB 6000
1990 RETURN
1999 REM*********************
2000 REM*  PUT CRSR @ R,C    *
2001 REM*********************
2050 PRINT LEFT$(P$,R+1)TAB(C);
2070 RETURN
2099 REM*********************
2100 REM*  PRINT Z$ @ R,C    *
2101 REM*********************
2150 PRINT LEFT$(P$,R+1)TAB(C)Z$;
2170 RETURN
2499 REM*********************
2500 REM* FLASH CRSR @ RP,CP*
2501 REM*********************
2520 CF=1+BD(RP,CP):R=RP+RO:C=CP+CO
2540 FOR LP=0 TO 1 STEP 0
2560 FOR CS=1 TO 2:DE=10:GOSUB 2800
2580 GET GT$
2600 IF GT$<>"" THEN LP=1:CS=2
2620 Z$=C$(CF,CS):GOSUB 2100
2640 DE=10:GOSUB 2800
2660 NEXT CS,LP:RETURN
2799 REM*********************
2800 REM*   DELAY FOR DE     *
2801 REM*********************
2820 FOR NN=1 TO DE:NEXT:RETURN
2999 REM*********************
3000 REM*  VALIDATE  INPUT   *
3001 REM*********************
3020 IF GT$="!" THEN R=18:C=0:
     GOSUB2000:STOP
3040 GT=ASC(GT$)-132:F=2*INT(GT/2)
3060 IF (GT<1)OR(GT>8) THEN PT=0:
     RETURN
3080 IF GT<5 THEN PT=1:RETURN
3100 PT=GT-3
3490 RETURN
3499 REMM*********************
3500 REM*  MOVE THE CURSOR   *
3501 REM*********************
3520 NY=RP+OF(2,GT):NX=CP+OF(1,GT)
3540 IF (NY<1)OR(NY>RL) THEN RETURN
3560 IF (NX<1)OR(NX>CL) THEN RETURN
3580 RP=NY:CP=NX
3620 RETURN
3999 REMM*********************
4000 REM*  TOGGLE A CELL     *
4001 REM*********************
4020 TG=1-BD(RP,CP):Z$=C$(1+TG,2)
4040 R=RO+RP:C=CO+CP:GOSUB2100
4060 BD(RP,CP)=TG:MP=NCGEN+CN*8-1
4120 PE=PEEK(MP+RP)
4140 PE=PE+(TG*2-1)*(2^(8-CP))
4160 POKE(MP+RP),PE:GOSUB6500:
     RETURN
4499 REMM*********************
4500 REM*  DEFINE NEW CHAR   *
4501 REM*********************
4520 FOR K=1 TO 1
4540 Z$=R$+"   CHANGE  CHAR    "
4550 R=14:C=7:GOSUB2100
4560 Z$=" NEW NUMBER        "
4570 R=15:GOSUB2100
4580 C=19:GOSUB2000:INPUT A$
4600 CN=VAL(A$):PRINT C2$
4620 IF (CN<0)OR(CN>127) THEN K=0
4640 NEXT K:GOSUB 6000
4660 Z$="                    ":C=7
4670 R=14:GOSUB2100:R=15:GOSUB2100
4680 RETURN
5999 REM*********************
6000 REM*    DISPLAY CHAR    *
6001 REM*********************
6020 MP=NCGEN+CN*8-1
6040 FOR RP=1 TO 8:PE=PEEK(MP+RP):
     Z$=""
6060 FOR CP=8 TO 1 STEP-1:
     N=INT(PE/2)
6080 Q=PE-2*N:BD(RP,CP)=Q:PE=N
6100 Z$=C$(Q+1,2)+Z$:NEXT CP
6120 R=RO+RP:C=CO+1:GOSUB2100
6130 NEXT RP
6140 X$=CHR$(CN+CX(INT(CN/32)))
6150 Z$=R$+"CHARACTER":R=1:C=11
6160 GOSUB2100:R=R+2:GOSUB2100
6170 Z$=" "+X$+" =      "
6180 R=R-1:GOSUB2100
6190 C=C+4:Z$=STR$(CN)+N$:GOSUB2100
6220 RP=1:CP=1
6490 GOSUB6500:RETURN
6499 REM*********************
6500 REM*    DISPLAY BYTES   *
6501 REM*********************
6540 C=CO+CL+2:FOR R=RO+1 TO RO+8
6560 Z$=STR$(PEEK(MP+R-RO))+"  "
6580 GOSUB2100:NEXT:RETURN
6999 REM*********************
7000 REM*   PLACE A CHAR     *
7001 REM*********************
7020 Z$=X$:C=C+OF:GOSUB2100:RETURN
60999 REM*********************
61000 REM* RELOCATE CHARGEN   *
61001 REM*********************
61100 CGEN=53248:NCGEN=14336
61120 ITRPT=56334:IOPT=1:PO=53272
61125 RETURN
61150 POKE IT,PEEK(IT)AND254
61200 POKE IO,PEEK(IO)AND251
61250 FOR J=0 TO 2047
61300 POKE (NCGEN+J),PEEK(CGEN+J)
61350 NEXT J
61400 POKEIO,PEEK(IO)OR4
61450 POKE IT,PEEK(IT)OR1
61500 POKE PO,(PEEK(PO)AND240)OR14
61990 RETURN
```

SWITCHED ON

Our practical course continues with explanations of how to build an interface device that enables your computer to control small electric motors and light bulbs.

IAN McKINNELL

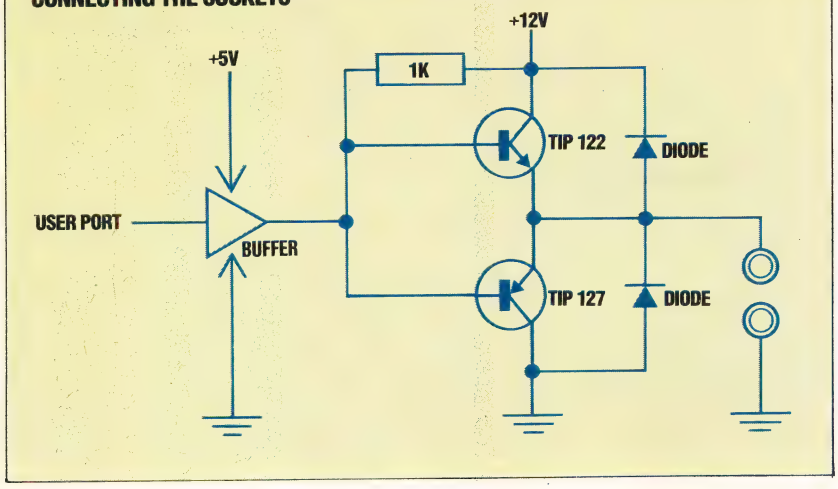
The Box
This is the finished box, showing the minicon plug and input/output leads. Care must be taken in cutting the board and the slot in the box so that the board does not move in the box when the plug is inserted into a connector

This interface will enable your home computer to control low voltage devices that take small power loads. The output from each of the four bits of the computer user port is buffered by the same buffer chip that we used in the construction of the buffer box in a previous instalment (see page 523). The output from this is used to switch the transistors, which are capable of controlling higher voltages and power than the buffer can handle. The complementary transistor configuration allows the output to both power a load or to sink current from a load. The two diodes at the output protect the transistors from reverse currents that some inductive loads, such as relays and motors, can create.

The interface can be used as a bi-directional motor controller. To simply switch it on and off, a motor is situated between the output connector and the earth connector. When a one is outputted from the computer, the output is set high and the motor starts. A zero output will turn the motor off.

Connecting the motor between two of the interface's outputs, however, will allow you to control the direction of the motor's movement. If the computer sends the same outputs (both zeros or both ones) then the same voltage will appear on both interface outputs and no current will flow through the motor. A one on one output and a zero on the other will give a difference in voltages that will make the motor turn in one direction. Reversing the logic will cause the current to flow — and the motor to turn — in the opposite direction.

CONNECTING THE SOCKETS



LIZ DIXON

Building The Interface

First of all, build the box that the interface will be cased in. The circuit board will fit exactly into the box specified for the buffer box project. The box must be drilled to accept the sockets and the bus plug (and minicon socket if required).

Once the sockets have been fixed in the box, then we have to make the connections between them. Using a piece of the tinned wire, connect all of the earth (black) sockets together. Then take a six inch piece of nine-way ribbon cable, and attach one strand to the tinned wire connecting the earth sockets. Attach the other eight strands, in twos, to each of the four output (red) sockets.

Now cut the veroboard to the right size (45 holes × 16 strips) and remove the half row of holes at one end. Keep this offcut from the board, as this will be used in constructing the other interface. Now make the track cuts as shown in photograph 'A'.

Solder the passive components in first: the chip socket, the bus socket and the wire links. If you want to fit the bus expansion socket, then fit this now, but leave the cable connecting this and the bus plug until last.

The resistors are soldered on next, followed by the diodes. Check to make sure that these are fitted the correct way round. Solder the eight transistors on next, the correct way round. Finally, solder the connections to the sockets as shown in photograph 'B', and fit the chip in position. You can now assemble the board in the box, and the interface is ready to use.

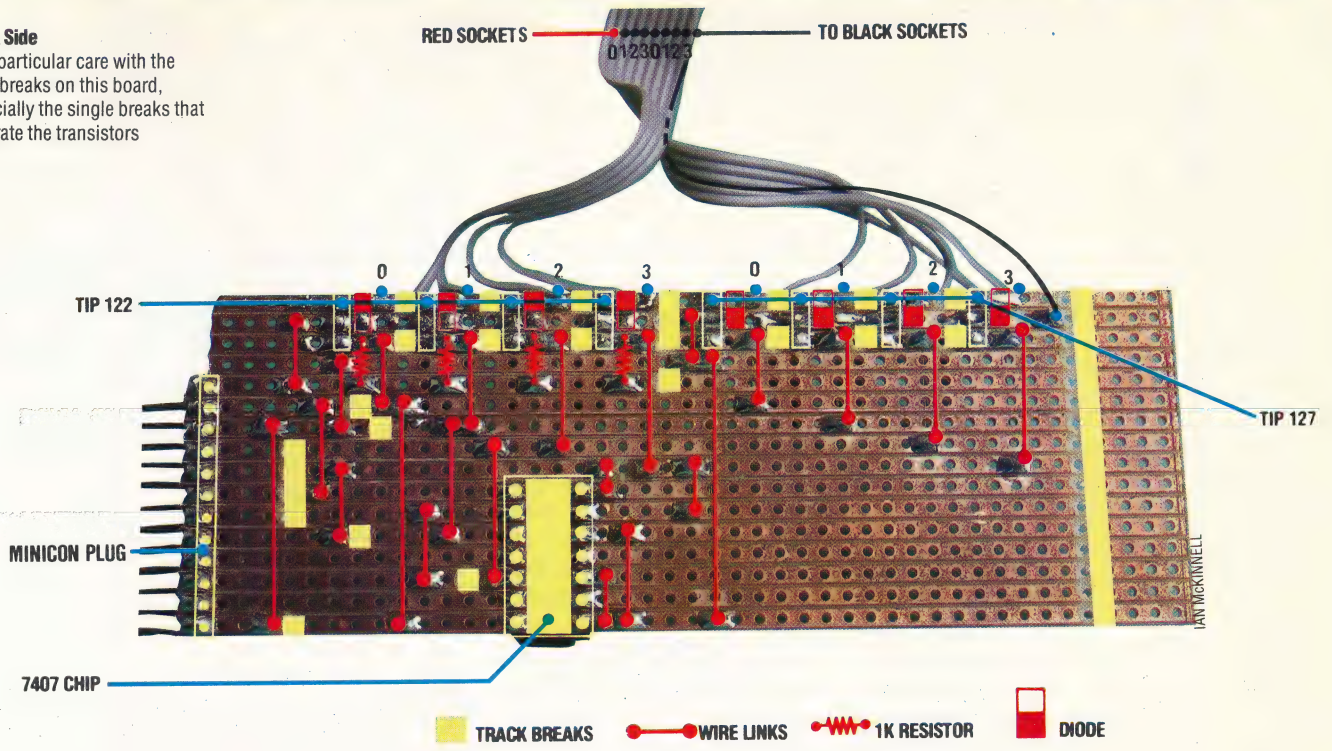
Parts List

Quantity	Item	Maplin No
1	50 hole x 24 strip veroboard	FL07H
4	1 K-ohm resistor	M1K
8	1N4001 diode	QL73Q
4	TIP 122 transistor	WQ73Q
4	TIP 127 transistor	WQ74R
1	7407	QX76H
1	14 pin DIL socket	BL18U
4	Red 4mm socket	HF73Q
4	Black 4mm socket	HF69A
4	Red 4mm plug	HF66W
4	Black 4mm plug	HF62S
1	12-way minicon plug	YW19V
1	120x65x40 mm '2004' box	LH60Q
1	12-way minicon socket	VW30H*

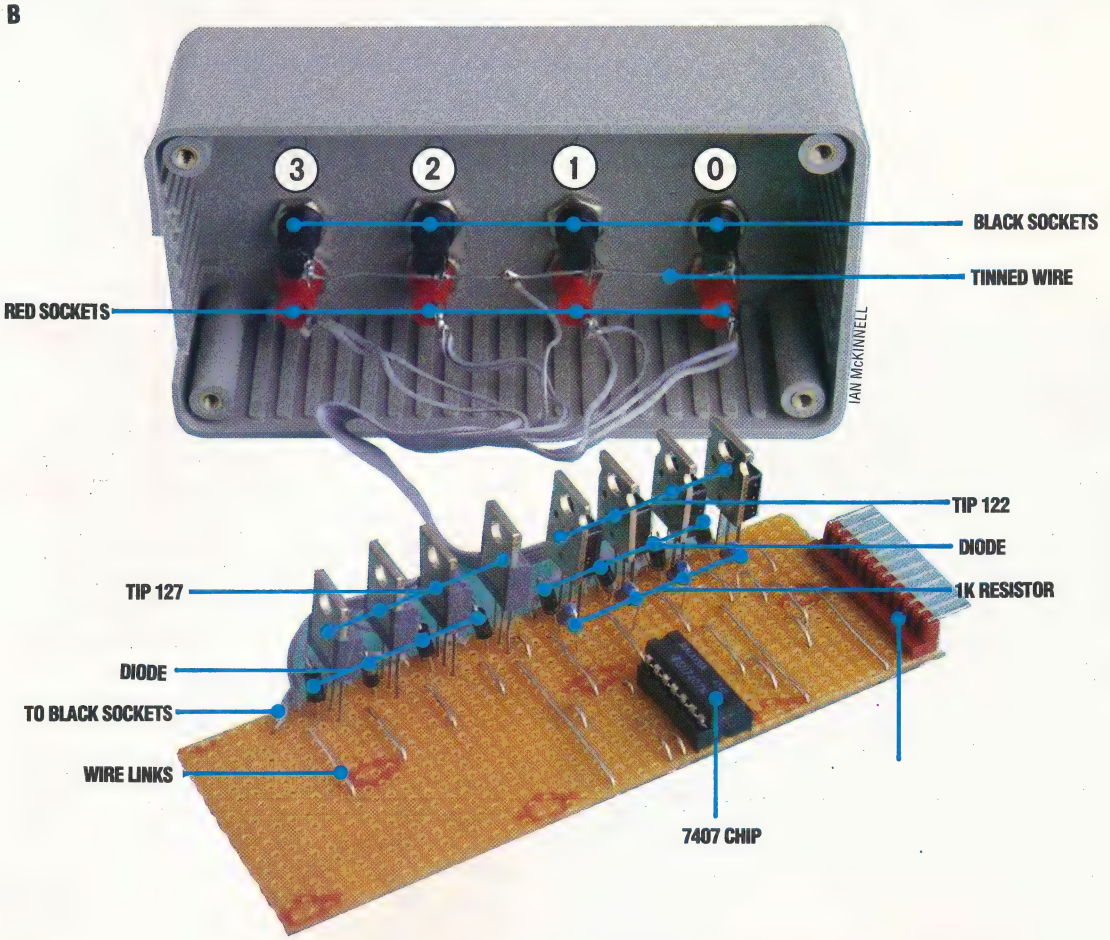
*The last item is optional. It extends the system bus beyond this interface so that further interfaces can be plugged in simultaneously.
You should have some plain tinned wire and some ribbon cable left over from the previous project.

Track Side
Take particular care with the track breaks on this board, especially the single breaks that separate the transistors

A



B



Component Side
The TIP 122s are at the right-hand end of the board, and the four diodes among them are connected with the black end to the edge track and the silver end to the next track. The ribbon cable line from the black sockets is soldered to the left-hand end of this track. The other four diodes are connected in the opposite sense among the TIP 127s. The minicon plug can be clearly seen on one end of the board

FIBONACCI SEQUENCE

Developed by a Florentine mathematician, Leonardo Fibonacci, in the 13th century, the *Fibonacci sequence* is an infinite series of numbers in which each number is the sum of the two preceding numbers. The sequence begins with 0, 1 and continues:

1 (the sum of 0 and 1), 2, 3, 5, 8, 13 . . .

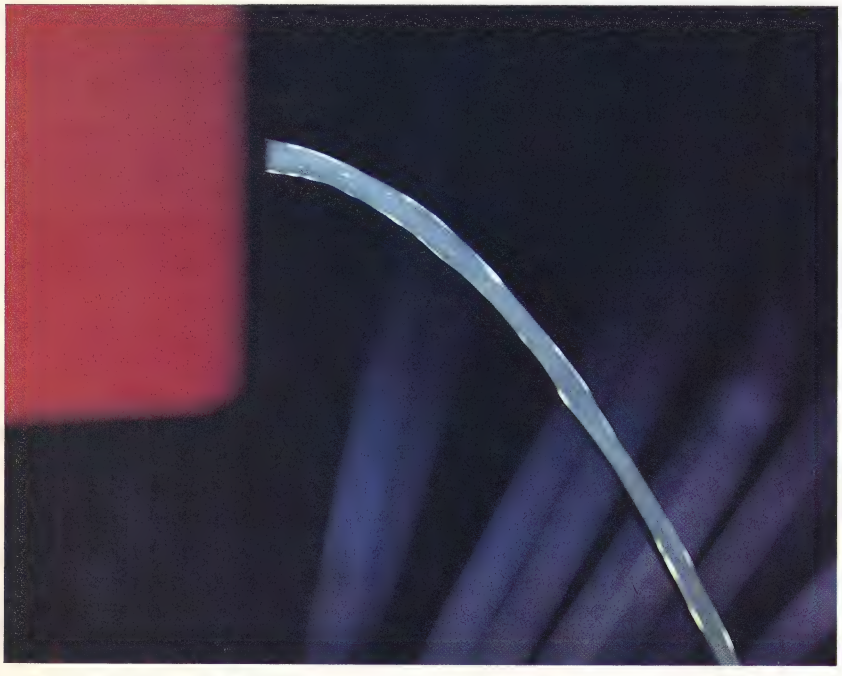
The formal mathematical definition of the Fibonacci sequence is:

$$F_0=0,\ F_1=1,\ F_{n+2}=F_{n+1}+F_n,\ n\geq 0$$

FIBRE OPTICS

Optical fibres, made of extremely thin glass or plastic, can be used instead of copper wire for voice or data transmissions over very long distances. A single strand of glass may carry several thousand signals. *Fibre optics* works on the principle of internal reflection: light is held within the strand because it reflects inwards from the exterior surfaces. This means that the strands may be bent or twisted through sharp angles with no effect on transmission. We can illustrate this principle by considering a container holding water. A light source shines into the container and, because the walls are solid, the light remains inside until a hole is created in one wall, allowing water to flow out. The water carries the light with it in the same way that an optical fibre would, so the light actually bends as it flows with the water.

The amount of information a fibre can carry and the quality of the signal transmitted both depend on the optical density of the glass. Cables constructed of optical fibres are non-conductive, which makes them useful in applications where normal conductive cables could present a safety hazard. They are also relatively secure, being much more difficult to tap into than ordinary coaxial cables.



NICK DALY

FIELD

In a database, a *field* is a group of data items under a specific heading. In a telephone directory, for instance, the surname is usually the first field, followed by the forename field, the address field, and the phone number field. A collection of fields comprising a defined range of information — a complete telephone book listing, for example — is called a *record*, while each individual piece of data (a specific person's surname, for instance) is an *entry*. The number of characters contained in an entry is often restricted; the length being determined by the nature of the field — so a surname field might be limited to 26 characters, the telephone number restricted to 10, and so on.

The word 'field' can also be used to refer to a descriptive element attached to a word or function. For example, in the address:

```
POKE 57367, n
```

the number 57367 can be referred to as the instruction field.

FIFO

An acronym for 'First In First Out', FIFO is one way of dealing with information held in a *stack* (a sequential data list in memory). The first element that is placed in the stack is also the first to be removed and acted on when the stack is filled. The stack may be manipulated in the opposite way, in which case it is referred to as LIFO (for 'Last In First Out'). A FIFO list is also called a *queue list* or a *pushup* stack.

FILE

A *file* is a collection of related information that is saved, altered and re-used after its creation. Computer files are saved to cassette or disk. They can contain whole programs, listings of programming instructions (often-used subroutines saved as library modules), data files that are loaded into other programs for specific applications, text files (such as memos, letters and other word processing documents), or graphic data for visual presentations.

In a database, information related to a specific subject forms a file, which is acted on as a unit. For instance, in a database of company records all information concerning personnel would be held in a 'Personnel' file. To locate information about one particular employee, the database would search the Personnel file, which would have its own location on the storage disk or cassette.

FILE MAINTENANCE

Several operations must be carried out on files to safeguard the information they contain and to ensure that such data is up to date. Making back-up copies of files, deleting out-of-date and unused records, and updating files are all part of *file maintenance*. Vital in a business or programming context, file maintenance is also essential in small databases, such as name and address files or a list of valuable items in a collection.

SIMPLE ARITHMETIC

It is now possible for us to take a detailed look at several machine code programs that show how some simple arithmetic is performed using the 6809 instruction set. We pay particular attention to signed arithmetic, and the use of the condition code register.

At this stage in the course, we can put some instructions together into a working program, although we will need to examine some new instructions and ways of representing data first of all. We will begin by devising a simple program that converts a binary coded decimal (BCD) number into its binary representation.

A binary coded decimal number (see page 168) is a way of representing a decimal number in binary form that is particularly useful when dealing with eight-bit processors. Using this representation, each digit in a decimal number is translated into its binary equivalent. The decimal number 69, for example, is equivalent to the BCD representation %01101001: the leftmost four bits (0110) are the binary equivalent of 6, and the rightmost four bits (1001) are equal to the decimal 9. Thus, using BCD, we get an entirely different decimal equivalent than we would if we were converting the *binary* number %01101001 (it is equivalent to 105 decimal).

Our conversion program will need a number of new instructions; let's consider these in turn:

- LSR (Logical Shift Right): This shifts every bit of the operand one place to the right. The rightmost bit is shifted into the carry bit of the condition code register of the processor, and a zero is shifted into the leftmost bit of the operand.

- AND: This logically ANDs each bit of a register with the corresponding bits of the operand, leaving the result in the register. This instruction is most often used to *mask* certain bits: if a register contains a one in a bit, then ANDing it with another bit will copy that second bit into the register; if the register bit contains a zero, then ANDing it will always result in a zero. For example, the effect of ANDing a register value of %00001111 with a given memory location is to copy the rightmost four bits only of the location into the register. Thus:

```
%00001111 Register value
%10110110 AND memory location value
%00000110 Result in register
```

- MUL: This MULtiplies the contents of the A and B registers, leaving the result in the D register (the 16-bit register formed from A and B together). Very few other eight-bit processors support multiplication as an op-code.

- SWI (SoftWare Interrupt): This is a convenient way of terminating a machine code program, returning control to the operating system. We shall examine this instruction in more detail when we consider the interrupt system later in the course. Here is the BCD-to-binary program:

- Specify value in location counter:

```
ORG $1000
```

- Store BCD 58 in BCDNUM and reserve byte at BINNUM:

```
BCDNUM FCB %01011000
BINNUM RMB 1
```

- Load BCD 58 into the A register and mask the lower digit. Store that digit in BINNUM:

```
START LDA   BCDNUM
      ANDA  #%00001111
      STA   BINNUM
```

- Load BCD 58 into A accumulator and shift upper digit (leftmost four bits) rightwards:

```
SHIFT LSRA
      LSRA
      LSRA
      LSRA
```

- Load 10 (decimal) into the B register and multiply by the contents of A:

```
MULT LDB   #10
     MUL
```

- The result is 16 bits in the D register, but as this result cannot be greater than 90 ($10 \times 9 = 90$), only the lower byte of D is needed. The lower byte is in the B register — so add its contents to BINNUM and store the result:

```
ADDIT ADDB BINNUM
      STB  BINNUM
```

- Thus, we have the BCD number in BCDNUM and the binary equivalent stored in BINNUM. We can finally return to the operating system and end the source code:

```
RETURN SWI
END
```

TWO'S COMPLEMENT

The programming examples we have given so far in the course have all involved simple arithmetic, and we shall continue in this vein for a little while longer. Let's now look at the problem of *sign* — by which we mean positive and negative numbers.

Decimal	Binary
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

The most common method of representing negative numbers in a computer memory location or register is the form known as *two's complement* (see page 328). To obtain the two's complement of a binary number, we invert all the digits (change all the zeros to ones, and vice versa), and add one to the number. Thus, the two's complement of 0101 it 1011.

But how is this used to perform mathematics involving signed numbers? First of all, let's consider the range of numbers that can be represented: an eight-bit register can hold only 256 different bit patterns, which can be used to represent positive numbers in the range 0 to 255 *or* negative and positive numbers in the range -128 to 127. (A 16-bit register can hold values from 0 to 65535 *or* -32768 to 32767.) We give a table in the margin that shows how a four-bit binary representation is made for the decimal values from -7 to 7.

If you look at the table, you will notice that the negative numbers all have a one in the most significant (leftmost) bit position. Similarly, all the positive numbers have a zero in the most significant bit position.

As you can see from inspecting this four-bit table, we can define some basic properties for signed mathematics based on two's complement:

- The two's complement of a negative number gives its positive equivalent, and vice versa.

- The most significant bit is always zero for a positive number, and one for a negative number. This makes recognising whether a number is positive or negative very easy.

- The two's complement of zero is zero (1111 plus 1).

- Addition and subtraction can be carried out in the usual way, and any given answer will have the correct sign.

You might like to try a few simple addition and subtraction sums to verify the legitimacy of the last property. Multiplication, however, is more difficult when using signed numbers. The MUL instruction that we used in the BCD-to-Binary program at the start of this instalment treats the contents of the A and B registers as unsigned numbers. If we want to multiply two signed numbers then we must program it ourselves.

Anybody who has done any programming will realise that we are extremely limited in what we can do using the simple 'linear' programs that we have so far used in this course. We can only begin to do useful things by employing one of the basic forms of control structure:

- Selection: in which we choose between two different courses of action (like the IF statement in BASIC)

- Repetition: in which we repeat a sequence of operations:

1) while a certain condition remains true (the WHILE . . . WEND structure);
2) until a certain condition becomes true (REPEAT . . . UNTIL); or
3) a certain number of times (FOR . . . NEXT).

All of these structures depend on the ability to test a condition to see whether it is true or false, the most common sort of condition being whether a variable has a certain value or not. In Assembly language, we need to use these structures, and will therefore need to be able to test the values in registers. We can usually test directly for only two possibilities (whether a value is zero or not, and whether it is positive or negative). With extra instructions, however, it is possible to carry out other sorts of test.

CONDITION CODE REGISTER

These conditions are made available by using the condition code (CC) register, which we briefly mentioned earlier in the course (see page 537). This is an eight-bit register, but unlike the other 6809 registers, we are not interested in the value stored there. Rather, we are concerned with the state (1 or 0) of each of the eight bits individually. Five of the eight bits are devoted to conditions of the type we have been discussing, the other three are concerned with the handling of interrupts (which we will examine in detail later in the course). One of the five, H (the Half carry flag), is almost solely concerned with BCD arithmetic, and doesn't concern us at present. The remaining four, which are important at this stage, are:

- C: The Carry flag, which holds the carry digit (or borrow in the case of a subtraction) from the most significant bit after an arithmetic operation. It also has a useful function when we want to shift the contents of an accumulator along by one bit; some of the shift operations put the bit that is lost off the end into C. This bit, for example, could be used to test whether a number is odd or even by having the least significant bit shifted into it and tested. This is bit 0 (the least significant bit) in CC.

- V: The oVerflow flag, which is set to one whenever the result of an arithmetic operation is too large for the register that is supposed to contain it. This is bit 1 in CC.

- Z: The Zero flag, which is set to one when the contents of a register are zero. This is bit 2 in CC.

- N: The Negative flag, which is a copy of the most significant bit (the sign bit) of a number in a register; in other words it is set to one if the number is negative. This is bit 3 in CC.

It is one of the most difficult aspects of Assembly language programming to keep track of the state of the flags. Not every instruction will set the flags, and some flags are set depending on the contents of the accumulator while others can depend on other registers as well. The safest procedure is to test only on the values in an accumulator, and to

do this at the point where the required value appears, since it is difficult to ensure that the flags are not changed by any intervening instructions.

The flags are tested by means of 'branch' instructions, which are the low-level equivalent of the BASIC GOTO command. The 6809 uses relative (rather than absolute) branches almost exclusively. The difference is that a *relative* branch transfers control by so many bytes forward (or back), while an *absolute* branch transfers control to a specified address. The effect, however, is the same. It distinguishes between *short* branches, where the range is expressed in a single byte (-128 to 127), and *long* branches, which can go anywhere in memory. We will be using short branches only.

The 6809 has a large set of branch instructions, and we will introduce these as we need them. The following examples illustrate the instructions used to test and compare the values held in the accumulators and the use of the branch instructions to select and repeat procedures.

- ANDCC: It is not possible to load values directly into the condition code register, but it is good practice to set all the flags you need to zero before you start using them. The easiest way of doing this is by using the ANDCC instruction, which operates just like an AND command, using zeros as masks in the bit positions we want to use.

- SUB (SUBtract): The operand is subtracted from the accumulator, which sets the C, V, Z and N flags on the result. (The H flag is also set if the subtraction is eight-bit).

- CMP (CoMPare): This works in exactly the same way as SUB, except that the contents of the register are left unchanged. As in SUB, the C, V, Z, N (and possibly H) flags are set.

- BRA (the unconditional BRAnch): This is just like the BASIC GOTO command.

- BGT (Branch if Greater Than zero): This is a test for the signed numbers. The branch takes place if Z is zero (the number is non-zero). To allow for the fact that the sign bit may be incorrectly set if overflow has occurred, either N must be zero and V also zero (straighforward non-negative) or N must be one and V also one (incorrectly negative due to overflow). Other similar tests for signed numbers are BGE, BLT and BLE.

- BLO (Branch if LOwer than zero): This is an unsigned test, since it is pointless inspecting N with unsigned numbers. The branch occurs if the C flag is set, indicating a borrow after a subtraction. Similar unsigned tests are BLS, BHI and BHS.

- A program to find the larger of two signed eight-bit numbers stored in $3000 and $3001. The larger of the two numbers to be placed in $3002. First label the numbers:

```
NUM1    EQU     $3000
NUM2    EQU     $3001
ANS     EQU     $3002
        ORG     $1000
```

- The code begins: the condition code flags are set to zero and the first number is loaded. This is compared with the other number:

```
        ANDCC   #%11110000
        LDA     NUM1
        CMPA    NUM2
```

- If NUM1 is the larger, then the program branches to FINISH. Otherwise it loads the second number into the A register. Whichever number is in the register when FINISH is reached is then stored in ANS, and the program returns to the operating system and ENDs:

```
        BGT     FINISH
        LDA     NUM2
FINISH  STA     ANS
        SWI
        END
```

Original Directives
The differing effects that assembler directives and Assembly language statements have on the assembler's location counter and on the contents of memory can be seen in this example

Original Directives

```
LABEL   OP-CODE  OPERAND      LOCATION   MEMORY
FIELD   FIELD    FIELD        COUNTER    CONTENTS
*----------DEMONSTRATION-------------
RESET   EQU      $F100
INDEX   EQU      16
MASK1   EQU      %01101010
                              $0400      ??
                              $0400      ??
                              $0400      ??

        ORG      $1000
                              $1000      ??

CR      FCB      16           $1000      $10

MEMTOP  FDB      $7FFF        $1001      $7F
                              $1002      $FF

TABLE1  RMB      7            $1003      ??
                              $1004      ??
                              $1005      ??
                              $1006      ??
                              $1007      ??
                              $1008      ??
                              $1009      ??

ERRMSG  FCC      'ERROR       $100A      $45
                              $100B      $52
                              $100C      $52
                              $100D      $4F
                              $100E      $52

        CLRA                  $100F      $4F

        END                   $100F      ??

*-------SYMBOL TABLE----------------
RESET   F100   INDEX   0010   MASK1   006A
CR      1000   MEMTOP  1001   TABLE1  1003
ERRMSG  100A
```

No ORG has been issued, so the location address is the assembler's default setting. Note that location address is not affected by EQU, and the contents of memory are as yet undefined

This sets the location as specified, but memory contents remain undefined

FCB causes the operand to be stored in the byte addressed by the location counter

FDB causes two bytes to be initialised

RMB reserves 7 bytes (contents undefined) by incrementing the location counter by that number

The ASCII codes of the operand string are placed in memory by the FCC directive

At last — an Assembly language operation! There is no operand, so we have only one byte for the op-code

Another directive that does not affect the location counter

This is how the symbols used in the program would be stored in the assembler's workspace for its own reference during the Assembly

THE TECHNOCRATS

Special Line
Memotech began with specialised products for the Sinclair ZX81, such as the Memopak, shown here, which provided the ZX81 with an extra 32K of RAM

Memotech is a British company that established its name as a manufacturer of useful add-ons and peripherals for the Sinclair ZX81. Recently the company began producing its own range of home microcomputers — the stylishly designed MTX series.

Memotech was established as a result of the enormous public interest in Sinclair Research's first microcomputers. Despite the popularity of the ZX80 and ZX81, it was soon apparent that the machines were severely hampered by lack of memory, and a huge market for add-on memory boards was created.

The company's founders were both lecturers at Oxford University: Geoff Boyd lectured in metallurgy at Wilson College, and Robert Branton taught mathematics at Christ Church. The two men first met at a computer exhibition at the university in 1981, and decided to work together on add-ons for the ZX81. Their first product was a 16 Kbyte expansion board for the machine. This was later followed by a whole series of 'Memopaks', including 32 Kbyte and 64 Kbyte RAM packs, the high resolution graphics (HRG) pack, a spreadsheet analysis (Memocalc), a word processor (Memotext), Centronics and RS232 interfaces and a keyboard.

When Sinclair Research released the Spectrum in 1982, Memotech decided against producing a range of add-ons for the new machine. Instead, using the experience and expertise it had gained in producing the hardware for the ZX81, Memotech chose to concentrate its resources on designing and building its own machines. Tim Spencer, Memotech's sales and marketing manager, explains: 'We felt that the ZX81 was not going to last much longer, so we decided to build our own computer. We did have the technology, after all. But the ZX81 has lasted far longer than we expected, and our packs are still selling well.'

Memotech estimates that international sales of its Memopaks have exceeded 250,000 units. The packs, along with the MTX range of machines, are manufactured at the company's headquarters in Witney, Oxfordshire. The firm currently employs 110 people.

The MTX range was officially launched in February 1984, and the company claims sales of about 25,000 machines since then. Like the BBC Micro, the MTX comes in two models: the 32 Kbyte MTX500 and the 64 Kbyte MTX512. The machines use a Z80A microprocessor, and offer 16 colours in high resolution mode (256 × 192 pixels). The MTX BASIC is similar to BBC BASIC. The computer also features an on-board assembler/disassembler.

The computer can also be expanded to make use of Memotech's HRX graphics package. Starting with an unexpanded MTX500, the user can add disk drives and the three graphics controller boards: a 96-bit processor main controller board, a 'Frame Grabber' and a three-channel A/D converter. The resulting system is able to produce animations, picture composition and graphic design up to a full typesetting capacity. The system costs around £4,500.

Fruits Of Success
After the success of its ZX81 expansion products, Memotech advanced to new products. One such product is a line of computer peripherals, like the 80-column dot matrix printer, the DMX 80.

Memotech's latest venture is the MTX512, a 64K home and small business microcomputer. The MTX512 can store data on cassette, or on the optional floppy disk drive unit pictured here

Asked about the design philosophy behind the MTX range, Tim Spencer said: 'We are aiming at the more serious home user and the business market. The machines are not aimed at the games market, although of course you can play all the usual games on them.'

Because the MTX is capable of running CP/M, it can take advantage of the range of software available. However, the company is aware of a lack of cassette-based software that would give the machine greater appeal to the less serious home user. There are currently only about 40 different cassettes available for the MTX, and the company is actively encouraging the development of more programs. 'We have done quite a lot over the last few months,' Tim Spencer commented. 'We have close links with Continental Software, and PSS is writing for us.' There will also be a number of educational packages in the near future.

DATABASE

Here, courtesy of Motorola Inc., is the second and concluding part of the 6809 programmer's reference card.

Instruction	Forms	Addressing Modes: Immediate Op	Immediate ~	Immediate #	Direct Op	Direct ~	Direct #	Indexed[1] Op	Indexed[1] ~	Indexed[1] #	Extended Op	Extended ~	Extended #	Inherent Op	Inherent ~	Inherent #	Description	5 H	3 N	2 Z	1 V	0 C
LSL	LSLA													48	2	1	A, B, M: C b7 b0 ← 0	•	↕	↕	↕	↕
	LSLB													58	2	1		•	↕	↕	↕	↕
	LSL				08	6	2	68	6+	2+	78	7	3					•	↕	↕	↕	↕
LSR	LSRA													44	2	1	A, B, M: 0 → C b7 b0	•	0	↕	•	↕
	LSRB													54	2	1		•	0	↕	•	↕
	LSR				04	6	2	64	6+	2+	74		3					•	0	↕	•	↕
MUL														3D	11	1	A × B → D (Unsigned)	•	•	↕	•	9
NEG	NEGA													40	2	1	$\overline{A}$ + 1 → A	8	↕	↕	↕	↕
	NEGB													50	2	1	$\overline{B}$ + 1 → B	8	↕	↕	↕	↕
	NEG				00	6	2	60	6+	2+	70	7	3				$\overline{M}$ + 1 → M	8	↕	↕	↕	↕
NOP														12	2	1	No Operation	•	•	•	•	•
OR	ORA	8A	2	2	9A	4	2	AA	4+	2+	BA	5	3				A ∨ M → A	•	↕	↕	0	•
	ORB	CA	2	2	DA	4	2	EA	+	2+	FA	5	3				B ∨ M → B	•	↕	↕	0	•
	ORCC	1A	3	2													CC ∨ IMM → CC				7	
PSH	PSHS	34	5+[4]	2													Push Registers on S Stack	•	•	•	•	•
	PSHU	36	5+[4]	2													Push Registers on U Stack	•	•	•	•	•
PUL	PULS	35	5+[4]	2													Pull Registers from S Stack	•	•	•	•	•
	PULU	37	5+[4]	2													Pull Registers from U Stack	•	•	•	•	•
ROL	ROLA													49	2	1	A, B, M: C b7 b0	•	↕	↕	↕	↕
	ROLB													59	2	1		•	↕	↕	↕	↕
	ROL				09	6	2	69	6+	2+	79	7	3					•	↕	↕	↕	↕
ROR	RORA													46	2	1	A, B, M: C b7 b0	•	↕	↕	•	↕
	RORB													56	2	1		•	↕	↕	•	↕
	ROR				06	6	2	66	6+	2+	76	7	3					•	↕	↕	•	↕
RTI														3B	6/15	1	Return From Interrupt					7
RTS														39	5	1	Return from Subroutine	•	•	•	•	•
SBC	SBCA	82	2	2	92	4	2	A2	4+	2+	B2	5	3				A − M − C → A	8	↕	↕	↕	↕
	SBCB	C2	2	2	D2	4	2	E2	4+	2+	F2	5	3				B − M − C → B	8	↕	↕	↕	↕
SEX														1D	2	1	Sign Extend B into A	•	↕	↕	0	•
ST	STA				97	4	2	A7	4+	2+	B7	5	3				A → M	•	↕	↕	0	•
	STB				D7	4	2	E7	4+	2+	F7	5	3				B → M	•	↕	↕	0	•
	STD				DD	5	2	ED	5+	2+	FD	6	3				D → M:M + 1	•	↕	↕	0	•
	STS				10 DF	6	3	10 EF	6+	3+	10 FF	7	4				S → M:M + 1	•	↕	↕	0	•
	STU				DF	5	2	EF	5+	2+	FF	6	3				U → M:M + 1	•	↕	↕	0	•
	STX				9F	5	2	AF	5+	2+	BF	6	3				X → M:M + 1	•	↕	↕	0	•
	STY				10 9F	6	3	10 AF	6+	3+	10 BF	7	4				Y → M:M + 1	•	↕	↕	0	•
SUB	SUBA	80	2	2	90	4	2	A0	4+	2+	B0	5	3				A − M → A	8	↕	↕	↕	↕
	SUBB	C0	2	2	D0	4	2	E0	4+	2+	F0	5	3				B − M → B	8	↕	↕	↕	↕
	SUBD	83	4	3	93	6	2	A3	6+	2+	B3	7	3				D − M:M + 1 → D	•	↕	↕	↕	↕
SWI	SWI[6]													3F	19	1	Software Interrupt 1	•	•	•	•	•
	SWI2[6]													10 3F	20	2	Software Interrupt 2	•	•	•	•	•
	SWI3[6]													11 3F	20	1	Software Interrupt 3	•	•	•	•	•
SYNC														13	≥4	1	Synchronize to Interrupt	•	•	•	•	•
TFR	R1, R2	1F	6	2													R1 → R2[2]	•	•	•	•	•
TST	TSTA													4D	2	1	Test A	•	↕	↕	0	•
	TSTB													5D	2	1	Test B	•	↕	↕	0	•
	TST				0D	6	2	6D	6+	2+	7D	7	3				Test M	•	↕	↕	0	•

Legend:

OP	Operation Code (Hexadecimal)	$\overline{M}$	Complement of M	↕	Test and set if true, cleared otherwise
~	Number of MPU Cycles	→	Transfer Into	•	Not Affected
#	Number of Program Bytes	H	Half-carry (from bit 3)	CC	Condition Code Register
+	Arithmetic Plus	N	Negative (sign bit)	:	Concatenation
−	Arithmetic Minus	Z	Zero (Reset)	∨	Logical or
•	Multiply	V	Overflow, 2's complement	∧	Logical and
		C	Carry from ALU	⊻	Logical Exclusive or

Notes:

1. This column gives a base cycle and byte count. To obtain total count, add the values obtained from the INDEXED ADDRESSING MODE table, in Appendix F.
2. R1 and R2 may be any pair of 8 bit or any pair of 16 bit registers.
   The 8 bit registers are: A, B, CC, DP
   The 16 bit registers are: X, Y, U, S, D, PC
3. EA is the effective address.
4. The PSH and PUL instructions require 5 cycles plus 1 cycle for each **byte** pushed or pulled.
5. 5(6) means: 5 cycles if branch not taken, 6 cycles if taken (Branch instructions).
6. SWI sets I and F bits. SWI2 and SWI3 do not affect I and F.
7. Conditions Codes set as a direct result of the instruction.
8. Value of half-carry flag is undefined.
9. Special Case — Carry set if b7 is SET.

IAN McKINNELL